HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

Semester Thesis

# Code Generator for UML State Machines

Daniel Michel

HSR University of Applied Science Rapperswil
MRU Software and Systems
Advisor: Prof. Dr. Andreas Steffen

Rapperswil
January 24, 2011

**Abstract**

UML state machine diagrams are a widely used tool to model behavior. This is especially true for embedded and real-time software engineering projects. However, transforming complex state machines with multiple state nesting levels from the model to code is a tedious and error-prone task.

This thesis introduces a code generator approach that transforms state machine models to code automatically. In contrast to existing tools that are able to do that, however, the generator described here is completely open. This enables the generator to be adapted and extended to meet project-specific requirements.

The generated state machines are efficiently implemented and run on an established state machine execution framework, requiring only a minimum of memory to perform their task. Also, they are deterministic in time and are thus ready to be used in projects with hard real-time requirements.

The thesis first evaluates different execution frameworks and selects one as target environment. Then, an efficient implementation of state machines is discussed, which consequently is built up as reference implementation for the generator. After an introduction to the generator framework itself, the generator templates and the infrastructure surrounding the generator are presented. Finally, a chapter on the integration of the various tools describes how to enable rapid software development with the generator at hand.

# CONTENTS

CHAPTER

ONE

# INTRODUCTION

## 1.1 UML State Machines

The Unified Modeling Language, currently at version 2.3, defines three fundamental modeling approaches for behavior [OMG10]. One of them is the UML state machine diagram. State machine diagrams illustrate the possible states of a part of a system, as well as transitions that define under which circumstances the state changes.

### 1.1.1 History

In 1987, David Harel developed a state modeling formalism called statecharts, that significantly improved finite state machines (FSM) by introducing composite states [Harel87]. FSMs require a distinct state node for every state-defining parameter combination, eventually leading to a state and transition explosion. With composite states, however, common behavior can be represented in super states. This approach greatly improves the readability of complex state diagrams. Consequently, the OMG adopted an object-based variant of Harel statecharts. The terms *state machine* and *statechart* are thus often used interchangeably. In this thesis, both terms refer to the UML variant, unless stated otherwise.

## 1.2 Model-Driven Software Development

MDSD is a technique where software is primarily constructed by building a model of the software. Then, code generators are used to derive working code from the model.

This approach gained some popularity because it enables reasoning about the software on a more abstract, conceptual level. One of the key benefits is that extensive model checks can be performed on the model, which enables the enforcement of architectural rules and design guidelines. Also, multiple code sets can be derived from a single model, for example to generate configuration files, web service interface descriptions or test stubs, for example.

Furthermore, a model diagram is a much better basis for discussions than code. This simplifies communication in teams and creates the possibility to discuss diagrams with non-technical people, too.

CHAPTER

TWO

# REQUIREMENTS AND PROBLEM ANALYSIS

Before getting started with the frameworks, I think it is essential to formulate the most important design goals and requirements for the complete generator system. The purpose of this is to not get lost in details and be able to have some points that the different frameworks can be compared to.

## 2.1 Design Goals

The following points are principles that I find important in general. Since the code generator is a software system that will be used to build other software systems, I think defining these principles will help the users see which development scenarios I had in mind when building the generator.

1. Favor the generation of clean, object-oriented C++ code over minimized, machine-only readable generator output.
2. Favor modern techniques of dynamic and parametric polymorphism over macros and old C-style approaches.
3. Favor modular designs over monolithic all-in-one solutions.
4. Put an emphasis on testability.
5. Enable development in teams.

## 2.2 Requirements for a Code Generator System

### 2.2.1 Functional Requirements

#### Execution Framework

**FR-1** The execution framework must support the UML state machine specification or a subset thereof. At the minimum, the following concepts must be supported:

- Simple states
- Composite states
- Initial transitions
- Entry, exit and transition actions

- Transition guards
- Signal and time events
- Arbitrary event parameters

**FR-2** The execution framework must provide an active object implementation as specified in the UML. This enables the actual execution of multiple state machines.

**FR-3** The execution framework must provide event queues that can be used to store events while an active object is processing another event. These event queues must be thread-safe, because events may arrive at the queue from different active objects at the same time.

#### Code Generator

**FR-4** The code generator must produce valid C++ code for the input model. The semantics of the generated code must conform to the UML.

**FR-5** The code generator must provide a mechanism to preserve user code. It is not feasible to provide user code in the modeling tool, so the generator must make sure that the user code is not overwritten when the model is regenerated.

**FR-6** The generator must verify the model before generation. It must flag every violation of the UML specification as well as unimplemented features as an error.

### 2.2.2 Non-Functional Requirements

**NFR-1** The execution framework must not contradict the UML state machine semantics in any way.

**NFR-2** The operational behavior[1] of the execution framework must be deterministic. This implies that there must not be any memory allocations on the heap after system initialization.

**NFR-3** The execution framework must be portable to at least MSVC 8.0 and greater on Windows systems, and to GCC 4.3 and greater on Unix and Linux systems.

**NFR-4** The code generator must be implemented in a well-structured and clear way. For someone new to the system, it must be possible to make minor modifications to the generator after a day of investigation.

## 2.3 Problem Decomposition

### 2.3.1 Overview

The goal of this section is to provide a high-level overview of the different components in a complete code generator solution for UML state machines. Also, this overview will help deciding if an existing execution framework is suitable for the purposes of this thesis (see *Analysis of Existing Statechart Execution Frameworks*).

Figure *High-Level Code Generator Elements* outlines the basic components that are involved when generating code from a state machine model. First, the user creates a state machine model, most likely in an UML editor. This model is then fed into the code generator, which generates a C++ representation of

[1] In contrast to system initialization and startup, the operational phase of a system is the period in which the system must potentially meet real-time requirements.

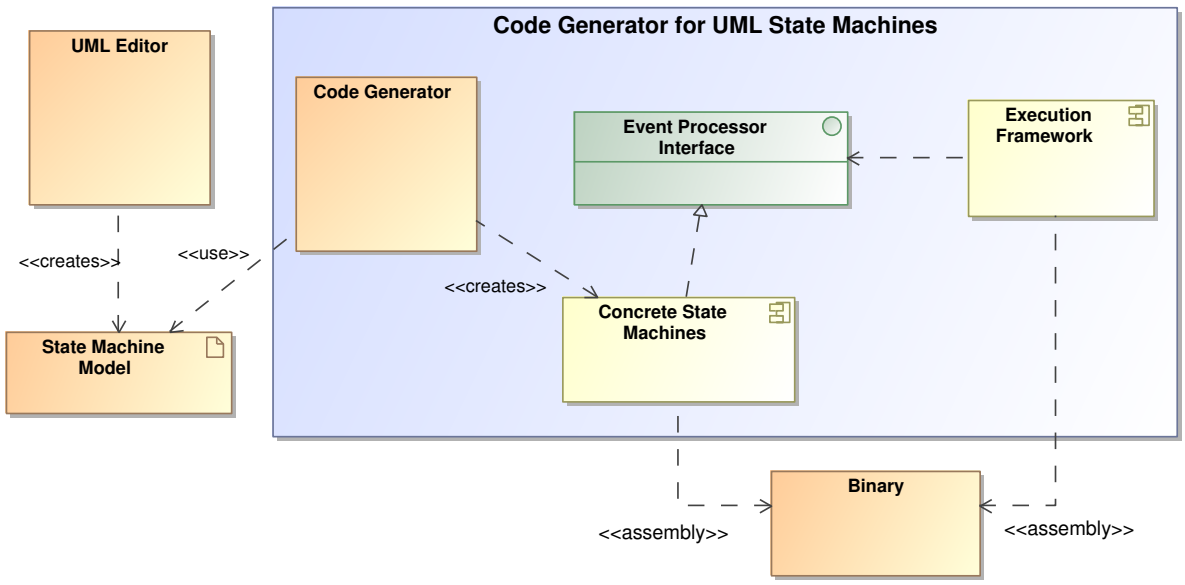


Figure 2.1: High-Level Code Generator Elements
From state machine design through code generation to the executable

the state machine that implements a previously defined event processor interface. Using this interface, an execution framework can collaborate with the state machines. Finally, these two components are linked into an executable binary[2]. The scope of this thesis is indicated by the blue system boundary rectangle.

### 2.3.2 Execution Framework

State machine execution frameworks may provide two basic services:

1. A hierarchical event processor that implements the UML state machine semantics.
2. An active object infrastructure containing event queues, memory pools, timers and/or other building blocks.

These two services are loosely coupled. Thus, when evaluating existing frameworks, one must be aware that it possibly implements just the first service. For a complete UML state machine solution, however, both are needed. The following definition is introduced to enable the categorization of execution frameworks:

**Definition 1**

**Complete Execution Framework** A state machine execution framework that provides a mechanism to express state machines as well as the infrastructure to execute them in a potentially multi-threaded environment.

**Event Processor Framework** A framework that just supports the specification of state machines and the enforcement of the UML semantics for these state machines.

[2] A real-world executable will certainly include many more components apart from the state machines and the execution framework.

Figure *Execution Framework Overview* shows a conceptual decomposition of execution frameworks. Note that the *Active Object* implementation and the *Event Processor* are only coupled by the *Event Processor Interface* and an *Event* interface that is not shown here. The concrete state machines interact with the framework through inheritance.

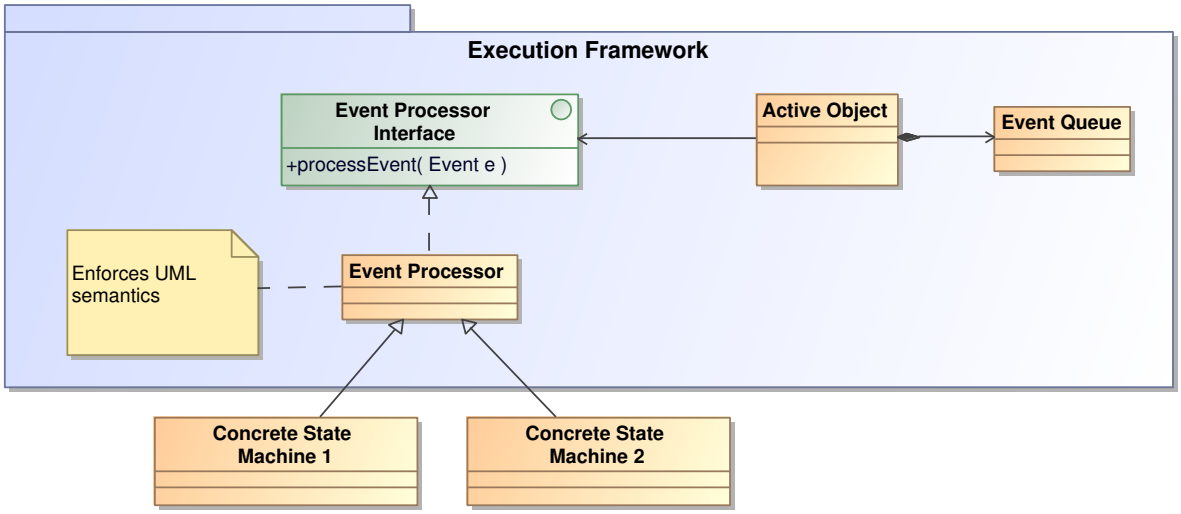


Figure 2.2: Execution Framework Overview

Theoretically, the following implementation strategies for an execution framework exist:

1. Take an existing framework that provides both an event processor and the execution infrastructure. The generator must then generate code specifically for that framework.
2. Take an existing framework for just one of the two services and build the other service from scratch.
3. Build the entire execution framework from scratch.

With these three basic possibilities in mind, the next step is now to investigate existing frameworks. This is what the following section is dedicated to.

CHAPTER

THREE

# STATE MACHINE FRAMEWORKS

## 3.1 Analysis of Existing Statechart Execution Frameworks

Several libraries or frameworks exist that help in developing statechart-based, event-driven applications. A survey was conducted to find potential state machine frameworks based on UML. It was carried out within the embedded and real-time division of Zühlke Engineering AG in Switzerland as well as on http://www.stackoverflow.com, a community-based Q&A website. The survey yielded the following candidate frameworks:

- **Boost Statechart**, a C++ template based approach to define state machines. Developed by Andreas Huber Dönni.
- **Boost Meta State Machine** (MSM), also a C++ template based approach, but with a state transition table as core element. Developed by Christophe Henry.
- **Quantum Platform** (QP), a C++ or C based state machine framework for embedded systems. Developed by Quantum Leaps.
- **Qt State Machine Framework**, an integrated part of the Qt cross-platform application and UI framework. Developed by Trolltech, now part of Nokia.

Note that frameworks that do not support hierarchical state nesting were not considered at all, since state nesting is a fundamental feature for this project.

### 3.1.1 Boost Statechart

Boost Statechart is a header only library that makes heavy use of template metaprogramming [Web-BoostStatechart]. As such, it implements the same basic idea as Boost MSM. Since Boost Statechart and Boost MSM are very similar and the time to analyse existing frameworks is limited, it becomes clear that evaluating them both does not make sense.

Boost MSM is preferred over Boost Statechart for this thesis due to the following reasons:

- After completing the Boost Statechart tutorial[1] it became clear that the syntax gets very confusing already with medium-sized state machines.
- Chapter 4 of Boost MSM contains a comparison of Boost MSM and Boost Statechart. It shows that MSM is considerably more efficient in terms of speed and size of the executable.

Thus, after completing the tutorial, Boost Statechart was not evaluated further.

[1] The Boost Statechart tutorial can be found at http://www.boost.org/doc/libs/1_44_0/libs/statechart/doc/tutorial.html.

### 3.1.2 Boost Meta State Machine

Boost MSM is also a header only, template metaprogramming based library [WebBoostMsm]. The version evaluated here is the one included with Boost 1.44.

MSM is split into a front-end, which provides the constructs to express state machines, and a back-end, which implements the processing according to UML. Note that there are three front-ends available. The eUML front-end is marked as experimental, so it is not considered a sensible choice for this thesis.

The fact that one front-end is experimental could lead to the impression that the library as a whole is still immature. This is not the case, however. Many mailing list entries show that Boost MSM is used in productive projects, some of them in the embedded and real-time domain. Also, the inclusion of MSM into Boost was judged by seven Boost reviewers who all voted for acceptance [MailAbrahamsMsm].

#### Transition Table Approach

In contrast to Boost Statechart, the central HSM definition element in Boost MSM is the transition table. Thus, after defining the states and action methods, the transition table is created to specify transition rows with the following information each:

- Source state
- Trigger event
- Target state
- Transition action
- Guard condition

While transition tables are a sensible approach for flat FSMs, they do not fit HSMs very well, because the state hierarchy is not visible. The state definitions that also define the hierarchy are separated from the transition table in the code, which reduces tangibility of the modeled behavior.

#### Readability of the Code

Even though Boost MSM took a lot of care to get clean and readable user code, it is still bloated. This is inevitable because of the template-based nature of Boost MSM, which makes `template` keywords and angled brackets ubiquitous. This is best seen in the simple tutorial[2] of the Boost MSM documentation.

#### Scalability

Scalability of Boost MSM is discussed in section 4 of [WebBoostMsm].

Defining large state machines in code seldom leads to readable, nice looking source files. But scalability is also an issue in terms of compilation time. This problem becomes even worse on certain compilers (e.g. MSVC 8.0, which is still widely used).

The runtime scalability in terms of time and space seems to be fine in Boost MSM, although no benchmarks were carried out.

[2] The implementation code of the Boost MSM simple tutorial can be found at http://www.boost.org/doc/libs/1_44_0/libs/msm/doc/HTML/examples/SimpleTutorial.cpp.

#### Conformance to the Project Requirements

According to *Definition 1*, Boost MSM implements an *Event Processor Framework*. As such, it helps only with expressing state machines and enforcing the UML semantics and fulfills only a subset of the *requirements*. This is only a small benefit for a generator solution compared to the increased complexity of the template based syntax that comes along with Boost MSM. After all, templates are also a kind of a code generator.

Also, there is currently a limitation that directly entering inner states of a composite state is not always possible. Although this does not violate *NFR-1*, it makes the generator considerably more complex, because a workaround consisting of explicit entries[3] must be implemented in the generator.

### 3.1.3 Quantum Platform

The Quantum Platform (QP) from Quantum Leaps, LLC is a state machine framework for embedded systems [WebQuantumLeaps]. It was developed by Miro Samek, who is also the president of Quantum Leaps and the author of "Practical UML Statecharts in C/C++" (see [Samek08]).

Two editions of QP are available; a C edition and a C++ edition, whereof the latter is considered here.

#### UML Conformance

QP is designed to be compliant to UML while only implementing a subset of the UML state machine specification [Samek08]. It supports hierarchical state nesting with entry, exit and transition actions. Also, nested initial transitions are processed in the correct way, and it is possible to guard transitions.

There is a set of rules that a developer must follow when defining state machines with QP. Mistakes that result from not following these rules are usually not detected by the framework and can cause undefined behavior. This is not so much a problem for this project, however, because the generator can be implemented in such a way that it never violates the QP rule set.

However, QP makes an profound trade-off: It accepts violating the action firing order in favor of a simpler implementation. More precisely, when exiting a state S through transition T, the exit action of S fires **after** the transition action of T. Thus, in this case, QP violates the UML and consequently *NFR-1*.

The example of a lamp that can simply be turned on and off demonstrates this behavior. It can be found in `Sources/Examples/QpLamp`. When executing the application, the output clearly shows the order in which the actions are executed:

```
1  LAMP DEMO
2  ~~~~~~~~~
3  Type 1 <Enter> to turn on the lamp, 0 <Enter> to turn it off
4
5  (Entry: Off)
6  The lamp is off
7
8  Command: 1
9  (Transition: Off->On)
10 (Exit: Off)
11 (Entry: On)
12 The lamp is on
```

[3] The limitation and the workaround are described in section 3 of the tutorial, see http://www.boost.org/doc/libs/1_44_0/libs/msm/doc/HTML/ch03s02.html#d0e1558

According to UML, the lines 9 and 10 should be swapped.

#### Scalability

With QP originating in the area of embedded and real-time systems, it is inherently designed to be scalable. From a software engineering point of view, this makes an exaggerated impression in some places. For example, the author avoided breaking up the event dispatcher function *QHsm::dispatch* into smaller functions to conserve stack space [Samek08]. Thus, the *dispatch* function ends up with a nesting level of ten and many, many lines of code.

But they did a good job in optimizing the framework: A small QP application, running on an RTOS, requires only about 10kB of program memory and 10kB of RAM [Samek08]. This shows that QP is optimized towards performance and small footprints.

#### Readability of the Code

With large state machines, as QP follows a code based approach, the code becomes also confusing. But, due to the state handler function approach, state machines stay clean longer than with Boost MSM. It also helps that the code for a given state, be it entry/exit actions, transitions or guards, is defined in the same state handler function. Thus, code belonging to a state is kept together and not spread across the source file.

#### Conformance to the Project Requirements

Regarding *Definition 1*, QP implements a *Complete Execution Framework*. In QP terminology, the event processor part is called *Quantum Event Processor (QEP)* and the execution framework is called *Quantum Framework (QF)*. QP is designed to be customizable in the sense that QEP or QF can independently be replaced with other implementations.

All execution framework specific *requirements* of this project are met, except for *NFR-1*, as stated above. Note, however, that the violation of *NFR-1* is specific to QEP and not QP in general.

### 3.1.4 Qt State Machine Framework

Qt started out as a cross-platform widget toolkit, but is now a complete application framework. Starting with version 4.6, Qt also features a state machine framework that builds upon Qt's event system [WebQtStateMachines].

#### Concepts

The Qt state machine framework implements all basic UML state machine concepts as required by *FR-1* as well as history pseudostates and parallel regions. It is based on established Qt concepts and makes extensive use of them: Events are Qt events, entry and exit actions can be connected to a state using the signals and slots mechanism, and transition triggers are also modeled as either Qt events or signals.

This makes the Qt state machine framework a very convenient solution if the state machines are to be used in a Qt application. For example, building GUIs and modeling their behavior with a state machine is straightforward.

But this ease of use is also the framework's weakness: Using the Qt state machine framework without using Qt itself is impossible. Employing Qt just as a glue layer to the state machine framework may work, but would not be feasible in terms of performance and maintainability.

Furthermore, using Qt state machines in a more customized way becomes cumbersome. Events with custom parameters, e.g., are bulky to use[4].

#### Performance

Since using the state machine framework without the rest of Qt is infeasible, the performance properties of Qt as a whole become relevant. Qt is a powerful, but at the same time heavyweight framework. The signals and slots mechanism, although being very flexible, is considerably slower than function calls [WebQtStateMachines]. And since every single action is connected to the state machine through signals and slots, this has substantial impact on the run-time performance.

Also, Qt is not suitable for very small systems. Hence, it covers a much smaller range of target platforms as the Quantum Platform.

#### Conformance to the Project Requirements

Qt is a *Complete Execution Framework* according to *Definition 1*. Using just the event processor is hardly possible due to the tight integration with the Qt core. The execution infrastructure, on the other hand, could be used with a custom event processor. However, this solution has the same drawbacks as using the built-in Qt event processor, because the whole system would again be tied to the Qt framework.

As a complete framework, the Qt state machines basically fulfill all execution framework specific *requirements*. Special care must be taken with the signals and slots mechanism not to violate *NFR-2*, however.

### 3.1.5 Evaluation Conclusion

In order to decide on the most suitable execution framework for this project, it is important to be aware that the state machine definition code is generated. This means that the constructs to define state machines in the code have no advantages for a generator solution. Thus, recalling *Definition 1*, *Event Processor* frameworks such as Boost MSM alone bring only the benefit of already implemented UML state machine semantics. As described above, there are considerable drawbacks of using Boost MSM, so it is judged unsuitable for this project. Its benefits are too small compared to the drawbacks.

The *Complete Execution Frameworks*, on the other hand, offer much functionality that is actually needed and can be reused. From the analysis above, it becomes clear that the Quantum Platform is superior to Qt for this project, except for the unacceptable violation of *NFR-1*. As a consequence, the following approach is chosen:

Reuse the active object infrastructure (QF) of the Quantum Platform without using its event processor (QEP). The benefits of having a custom event processor outmatch the increased development effort by far:

1. Obviously, the self-made event processor can be designed not to violate *NFR-1*.

[4] This can be seen in the example of the reference documentation, see http://doc.qt.nokia.com/4.7/statemachine-api.html#events-transitions-and-guards

2. Since the concrete state transitions are known at code generation time, the transitions can be hard-coded in the generated source file. Thus, the complicated and inefficient transition lookup algorithm in the event dispatcher of QEP is replaced by a faster and cleaner solution that is tailored to the concrete state machine.

   This approach is also known as *Model Ignorant Generation*, in contrast to *Model-Aware Generation* [Fowler10]. While the former generates working code directly, the latter splits the code into a generic and a specific part, with the generic part implemented in a library and the specific part generated.

3. The state machine source code can be organized in a way that combining the generator output with user-supplied code is straightforward for the user of the generator. This is possible because the structure of the state machines is not dictated by QP when using a self-made event processor.

### Example

In order to make it clearer why it is beneficial to replace QEP, an example is provided with both the QEP and the direct implementation, which corresponds to the generator output.

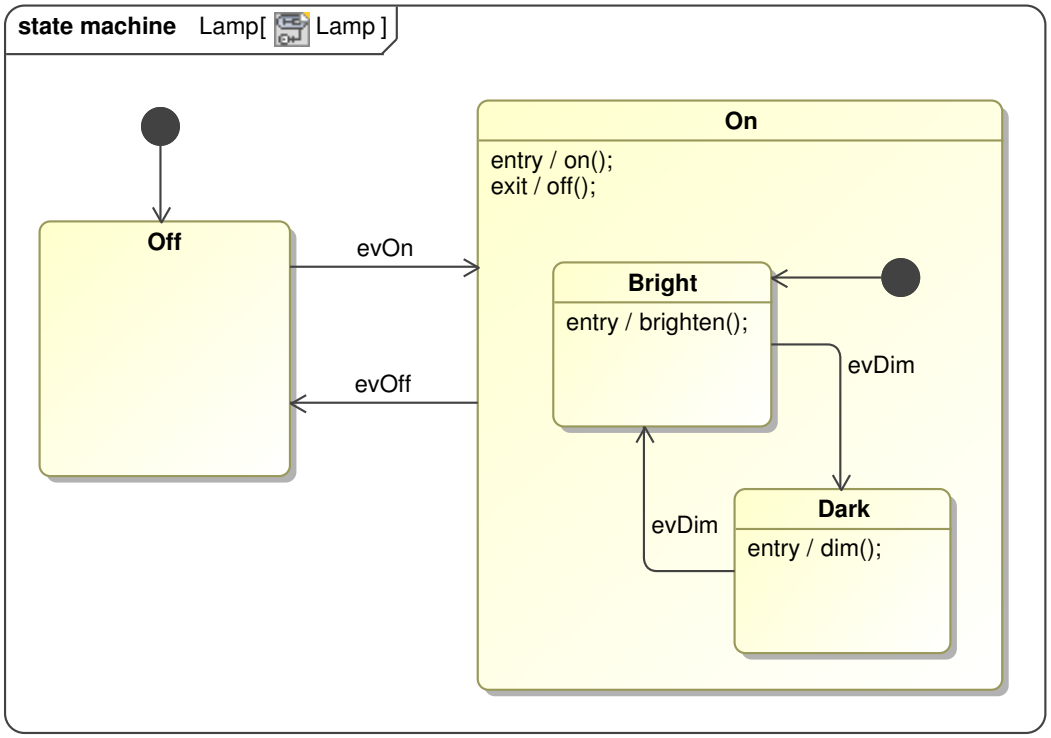


Figure 3.1: Lamp State Machine

Figure *Lamp State Machine* shows the state machine of a lamp that can be turned on and off with the respective events. When switched on, it can also be dimmed by sending a dim event.

When implementing this state machine for QEP, the state handler implementations would look as follows:

```
1 QState Lamp::Off ( Lamp * me, QEvent const * e ) {
2     switch(e->sig) {
3         case sigOn:
```

```
4               return Q_TRAN(&Lamp::Bright);
5       }
6       return Q_SUPER(&QHsm::top);
7   }
8
9   QState Lamp::On ( Lamp * me, QEvent const * e ) {
10      switch(e->sig) {
11          case Q_ENTRY_SIG:
12              me->on();
13              return Q_HANDLED();
14          case Q_EXIT_SIG:
15              me->off();
16              return Q_HANDLED();
17          case sigOff:
18              return Q_TRAN(&Lamp::Off);
19      }
20      return Q_SUPER(&QHsm::top);
21  }
22
23  QState Lamp::Bright ( Lamp * me, QEvent const * e ) {
24      switch(e->sig) {
25          case Q_ENTRY_SIG:
26              me->brighten();
27              return Q_HANDLED();
28          case sigDim:
29              return Q_TRAN(&Lamp::Dark);
30      }
31      return Q_SUPER(&Lamp::On);
32  }
33
34  QState Lamp::Dark ( Lamp * me, QEvent const * e ) {
35      switch(e->sig) {
36          case Q_ENTRY_SIG:
37              me->dim();
38              return Q_HANDLED();
39          case sigDim:
40              return Q_TRAN(&Lamp::Bright);
41      }
42      return Q_SUPER(&Lamp::On);
43  }
```

Here, we have a state handler function for each state, be it a simple or composite state. The `Q_SUPER` macro at the end of each handler is used to specify the state hierarchy. If an event arrives, the `QHsm` event processor proceeds as follows [Samek08]:

1. Call the current state handler and pass the event.
2. The state handler either does or does not handle the event:
   (a) The state handler returns `Q_HANDLED` or `Q_TRAN`. Thus, the event has been processed and the current RTC step terminates.
   (b) The state handler returns `Q_SUPER` that points to another state handler. Restart at (1.) with the returned parent state handler as the current state.
   (c) The state handler returns `Q_SUPER` that points to `QHsm::top`. This indicates that the event was not handled by any state handler in the hierarchy and is to be discarded.

The time complexity of this algorithm can be derived from two observations: With $n$ denoting the

number of different signals, each `switch` statement requires $O(log(n))$ steps to find the matching `case` in a typical implementation. If the event is not handled at all, every state handler in a hierarchy with depth $d$ must be called. Thus, the overall worst-case complexity of this algorithm is $O(d \cdot log(n))$.

Now let's take a look at the direct implementation, that is, how a generator could produce state machines:

```
1  void Lamp::enter_On() {
2      on();
3  }
4
5  void Lamp::exit_On() {
6      off();
7  }
8
9  void Lamp::enter_On_Bright() {
10     brighten();
11 }
12
13 void Lamp::enter_On_Dark() {
14     dim();
15 }
16
17 QState Lamp::Off ( Lamp * me, QEvent const * e ) {
18     switch(e->sig) {
19         case sigOn:
20             me->enter_On();
21             me->enter_On_Bright();
22             me->currentState = &Lamp::Bright;
23             break;
24     }
25 }
26
27 QState Lamp::Bright ( Lamp * me, QEvent const * e ) {
28     switch(e->sig) {
29         case sigOff:
30             me->exit_On();
31             me->currentState = &Lamp::Off;
32             break;
33         case sigDim:
34             me->enter_On_Dark();
35             me->currentState = &Lamp::Dark;
36             break;
37     }
38 }
39
40 QState Lamp::Dark ( Lamp * me, QEvent const * e ) {
41     switch(e->sig) {
42         case sigOff:
43             me->exit_On();
44             me->currentState = &Lamp::Off;
45             break;
46         case sigDim:
47             me->enter_On_Bright();
48             me->currentState = &Lamp::Bright;
49             break;
50     }
51 }
```

An important difference of this variant is that there are only state handlers for simple states, but not for composite states. This is because the state machine hierarchy is flattened in the code, and the actions of the composite states are included in all relevant sub state handlers (see `sigOff` handling in `Bright` and `Dark`, e.g.). Also, the approach is different: Rather than having the code for a specific state in one state handler, complete action sequences are grouped together.

In order to prevent user code like the `on();` and `off();` from getting duplicated, additional wrapper methods are introduced for each action of the original state machine.

The time complexity of this approach is obvious: Since there is no recursive state handler invocation, just the `switch` statement is relevant. Thus, the complexity is $O(log(n))$ in time.

## 3.2 Introduction to the Quantum Platform

After having decided on the state machine framework to be used in this project, the intent of this section is to provide a short introduction to the relevant parts of QP. Further details on design decisions and implementation details of QP, as well as an in-depth discussion of the source code, can be found in [Samek08].

As already pointed out earlier, event processing in QP is split up in two libraries: The hierarchical event processor, called QEP, and the active object execution framework, called QF.

### 3.2.1 Hierarchical Event Processor QEP

Even though QEP will not be needed in the generator solution (see *Evaluation Conclusion*), it is briefly discussed here anyway. This is due to the fact that the generated state machines must act as event processors when collaborating with QF. Thus, it is important to know which responsibilities and collaborations QEP has. Figure *QHsm Interface* shows the relevant interface of `QHsm`, which is the primary class of interest in QEP[5].

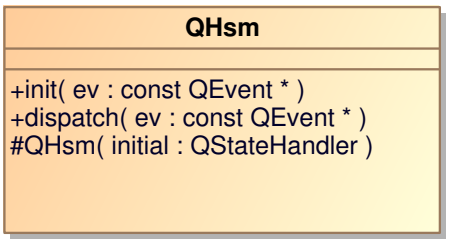


Figure 3.2: QHsm Interface
The hierarchical event processor class of the QEP library

The `init()` method is called by QF when an active object is started. This method instructs the state machine to recursively follow the initial transitions until a simple state[6] is found. The optional `ev` parameter can be used to pass initialization data in the form of "initialization events" to the state machine. This feature is not defined in the UML and is specific to QP.

[5] QEP contains also a non-hierachical state machine implementation called `QFsm`, but `QFsm` is not of interest for a generator solution.

[6] In UML terminology, a *simple state* is a state without any substates [OMG10]. Outside UML, they are sometimes also called *leaf states*. See *UML State Machine Metamodel* for a definition.

The `dispatch()` method is the main event processing interface. QF calls `dispatch()` when an event arrives or is waiting in the event queue. The event must then be handled according to the specification of the state machine. This method effectively executes a whole run to completion step.

The constructor of `QHsm` takes the initial state as a parameter and is protected to declare `QHsm` abstract. The concrete state machine must call the `QHsm` constructor in its own constructor and pass the initial pseudostate.

### 3.2.2 Active Object Execution Framework QF

The Quantum Framework QF forms the middle layer between the application and the OS. It provides the computation primitives that are required to execute state machines. Figure *Overview of the Quantum Framework* shows the central class of QF, `QActive`, and its relations in a QP application.

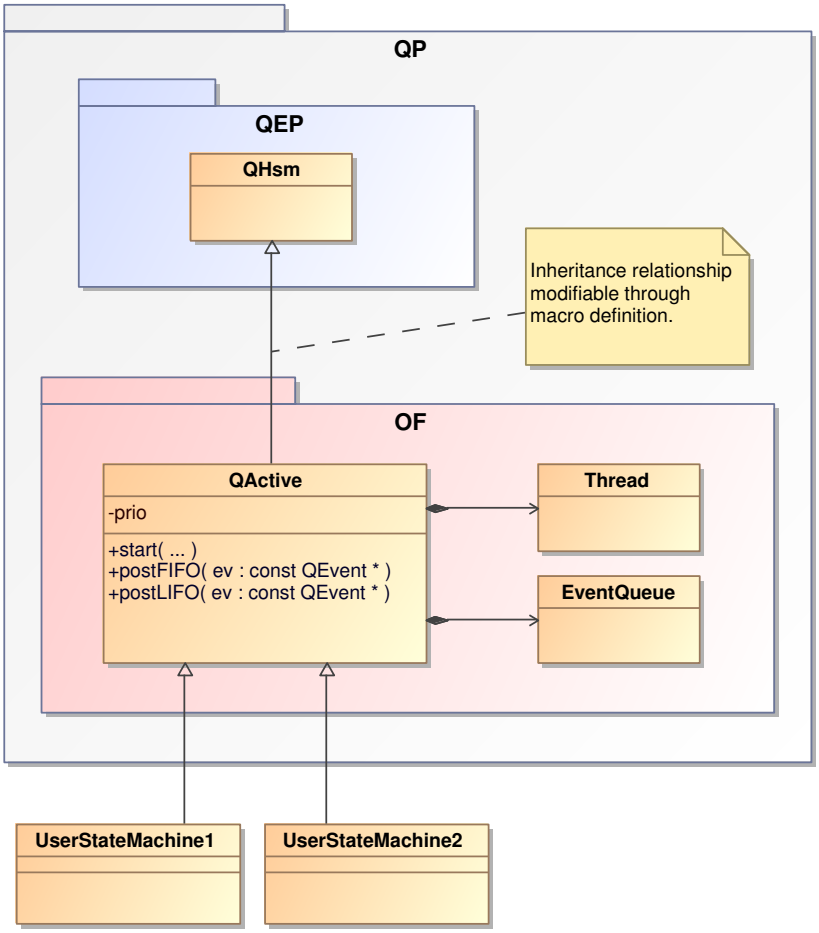


Figure 3.3: Overview of the Quantum Framework

Active objects are started by invoking `start()` on them. That method takes a whole bunch of parameters, with the important ones being the priority, space for the event queue, and the stack configuration of the thread the active object runs on.

`QActive` then takes care of starting a new thread through the platform specific portion of QF and sets up the event queue. Finally, the `init()` method is invoked to instruct the state machine to proceed to the initial state.

QF handles all the synchronization that is needed for the state machines to interact in a thread-safe manner. Thus, the user of the framework does normally not need any low-level synchronization primitives from the OS.

An event can be posted to a state machine by calling `postFIFO()` and passing the event. QF then either stores the event in the event queue or dispatches it directly to the state machine, depending on whether the state machine is currently busy or not.

There is also a `postLIFO()` method to store the event at the head of the event queue, thus outrunning all other events that are currently waiting in the queue. This should be rarely used, however, because it makes reasoning about state machines more difficult.

### 3.2.3 Lamp Example

In order to show the usage of QP (without modification), I implemented a small example. It consists of a lamp that can be switched on and off using some keyboard commands. Having the example ready in standalone-QP opens up the possibility to compare it to the generated version.

The example can be found in `Sources/Examples/QpLamp/`.

## 3.3 Quantum Platform Adaption Layer

As stated in section *Evaluation Conclusion*, the QF part of QP shall be reused. Since QF is normally used together with QEP, some mechanism is necessary to detach QEP and use QF with the generated state machines.

QP is designed to be portable to many different processor architectures and compilers. Thus, it makes sense to use this capability and create a new port to adapt QP to the needs of the generator. A guide on how to create new ports can be found in [Samek08].

The platform that is used for testing and also forms the first target platform is the Windows OS with use of the MSVC compiler. A port for this target already ships with QP (see folder `ports/80x86/win32/vc6/` in the QP directory). Thus, the existing port was copied and then modified to match the generator requirements[7]. It follows the QP port directory naming convention and is therefore structured in folders `custom_ports/80x86/win32/vc6gen/`.

The modified port can be found in the directory `Frameworks/GeneratorQpPort/` of the delivered project sources. The folder `custom_ports` must be copied to the QP root directory to include the port in QP.

### 3.3.1 QStateMachine

An important decision in the port is how to replace `QHsm`. Note that `QHsm` cannot just be left out, because `QActive` inherits from it.

The new base of `QActive` does not need to contain any implementation code anymore, since that moves to the generated state machines. It is therefore just an abstract class for state machines and has

[7] Modifying an existing port just by supplying some additional header files is not possible due to file reference clashes. A separate copy of the port, however, does not rely on the original port in any way.

been named QStateMachine[8].

The base class to be used by QActive can be controlled by the macro QF_ACTIVE_SUPER_. Obviously, this needs to be set to the abstract state machine class QStateMachine.

A second macro QF_ACTIVE_STATE_ specifies the type of a state handler and is used by QActive to define the type of the constructor parameter. This mechanism is used in QEP to pass the initial state to QHsm. However, the generator can generate the initialization code directly into the init() method of the concrete state machine. Consequently, the QF_ACTIVE_STATE_ macro is irrelevant, but must match the QStateMachine constructor anyway. Otherwise, the preprocessor would produce illegal code in QActive. Since it is irrelevant, the macro has been set to the primitive type int.

The following listing shows the important parts of the file qf_port_custom.h:

```
1  #include <qevent.h>
2
3  class QStateMachine {
4
5  public:
6      QStateMachine(int unused);
7      virtual ~QStateMachine();
8
9      virtual void init(QEvent const * e = 0) = 0;
10     virtual void dispatch(QEvent const * ev) = 0;
11 };
12
13 #define QF_ACTIVE_SUPER_ QStateMachine
14 #define QF_ACTIVE_STATE_ int
```

Note that the constructor does not need to be protected as in QHsm, because QStateMachine contains pure virtual methods and is therefore abstract by definition. The implementations of the constructor and destructor are in the file src/qf_port_custom.cpp and are both empty.

The file qf_port_custom.h is included in qf_port.h of the copied port to effectively get included by the concrete state machines. Apart from that include directive in qf_port.h, no other modification was made to the copied files. This approach minimizes QP upgrade hazards.

With the QP port developed in this section, the *Event Processor Interface* and *Execution Framework* components as identified in section *Problem Decomposition* are complete.

[8] Note that the "Q" has been kept in the name, even though this is a custom port and not part of the official QP distribution. However, the user of the generator does not care which class originated where, so the "Q" is used to distinguish user classes from the framework classes.

CHAPTER

FOUR

# UML STATE MACHINE METAMODEL

The code generator is backed with a metamodel that specifies the model element types and their relationships that are to be expected in a concrete model. It is thus important to have a thorough understanding of the metamodel when developing a generator.

Also, when talking about the reference implementation, it is beneficial to know about the metamodel in use. Thus, a introduction to the UML metamodel is provided early in this documentation.

The UML metamodel contains 247 metaclasses. For the state machine generator, however, we are only interested in a small subset of them, mostly those from sections 13 (Common Behaviors) and 15 (State Machines) of [OMG10].

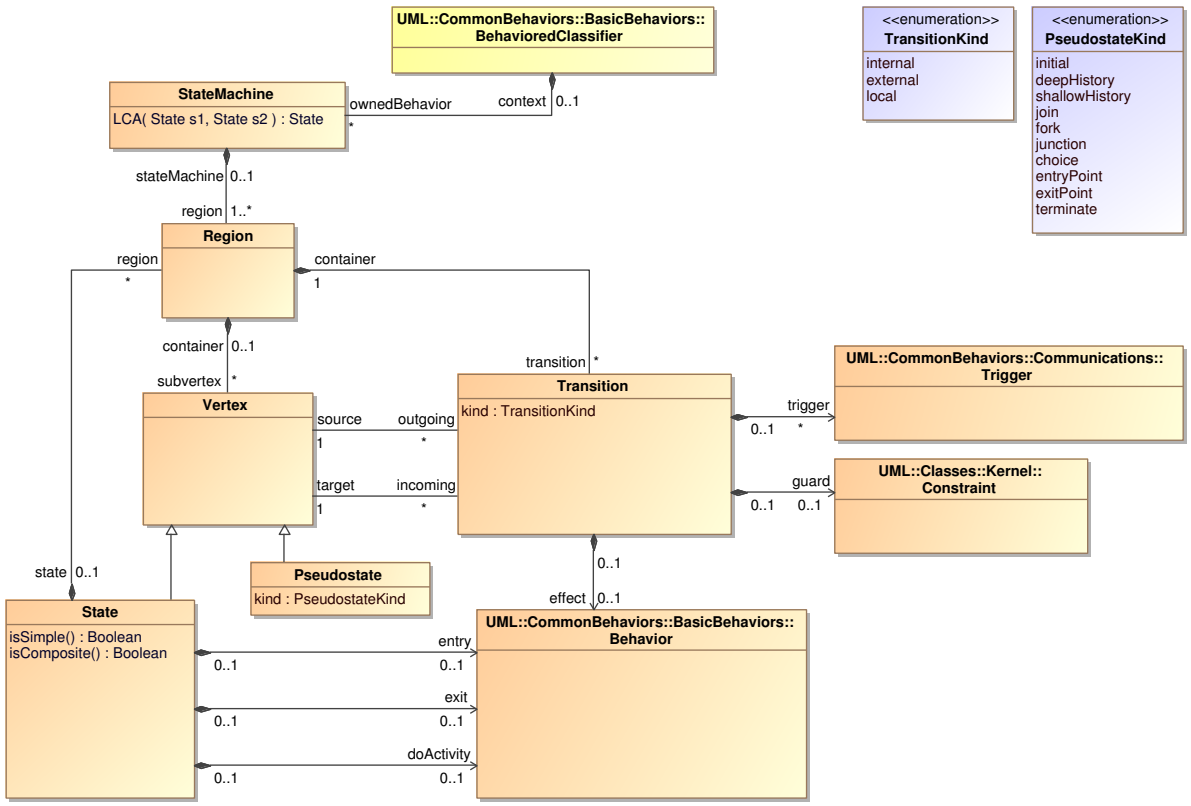


Figure 4.1: UML State Machine Metamodel

UML metaclasses and their relations, contained in UML::StateMachines::BehaviorStateMachines

Figure *UML State Machine Metamodel* is a simplified version of figure 15.2 from [OMG10] and shows the important metaclasses in the UML state machine package. In the following, I will introduce the

metaclasses and explain their usage in this thesis. See [OMG10] for the official specification of the metaclasses.

**StateMachine** The *StateMachine* is the root of all state machine specific elements and is contained in a *BehavioredClassifier*, which is typically a *Class*. A *StateMachine* contains one or more *Regions*, which contain the top-level states of the *StateMachine*.

**Region** A *Region* is the space inside a *State* or *StateMachine* and contains *Vertices* and *Transitions*. *Regions* are required in the metamodel because UML allows a *State* to contain more than one *Region*, so-called parallel *Regions*. Thus, *Vertices* cannot be contained directly in *States*.

**Vertex** *Vertex* is an abstract metaclass and forms the generalization of *State* and *Pseudostate*. It is an element that can be contained in *Regions* and may be targeted by *Transitions*.

**State** A *State* is a *Vertex* in which the modeled system may rest a while, in contrast to a *Pseudostate*. A run to completion step always takes the state machine from one *State* to another.

A *State* is either a *Simple State* or a *Composite State*. The former are the leafs in the state hierarchy tree, whereas the latter are the intermediate nodes that contain other *States* through a nested *Region*.

**Pseudostate** *Pseudostates* are *Vertices* of a transitional form, meaning that the state machine cannot stay in a *Pseudostate*. The different types of *Pseudostates* are distinguished by the `kind` attribute, which has the metatype *PseudostateKind*.

**Transition** A *Transition* is an arrow leading from a source *Vertex* to a target *Vertex*. If it is triggered, the source *State* will be left to enter another one.

*Transitions* may also be guarded, which means that there is a condition in the form of a *Constraint* attached. A guarded transition fires only if the guard evaluates to `true`.

Also, *Transitions* may have an action attached in the form of a *Behavior*, which is executed when the transition is being followed.

**Constraint** *Constraints* are generic constructs that impose a restriction on their context. Here, they are used to specify transition guards.

**Behavior** A *Behavior* is used in the context of state machines to specify entry and exit actions of states, as well as transition actions.

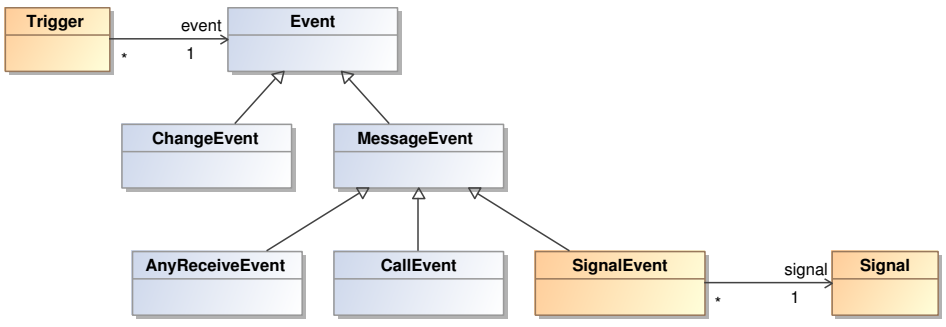


Figure 4.2: UML Event Metamodel

A merge of figures 13.11 and 13.12 from [OMG10]. The unimportant metaclasses are shown in gray.

As can be seen in figure *UML State Machine Metamodel*, *Triggers* are not specified in the state machine package, but in package "Common Behaviors", which also contains the metaclasses *SignalEvent* and *Signal*. Figure *UML Event Metamodel* shows the metaclasses in that package that are relevant here.

**Trigger** *Triggers* are the elements that link *Transitions* to *Events*. Apart from that connection, they contain no further information that is of interest for state machines.

**SignalEvent** Of the many types of *Events* in UML, only the *SignalEvents* are currently implemented in the generator.

**Signal** The *Signal* that is attached to every *SignalEvent* specifies the type of event occurrence, it is the "essence" of the *SignalEvent*.

At first, the distinction between *Signals* and *SignalEvents* can be quite confusing, but they indeed represent different concepts: An *Event* models an instance of an occurrence, whereas a *Signal* models the type of the occurrence. If, for example, the "up" button on an elevator is pressed three times, this produces three events. However, these events all have a single signal, namely the "up" signal.

CHAPTER

FIVE

# REFERENCE IMPLEMENTATION

When designing a software system that generates code, it is usually a good idea to create a reference implementation (RI) of the code that is later to be generated. This way, the problems of creating a runnable program that does what it is supposed to is separated from the problem of implementing a generator that produces exactly that code.

Of course, when implementing the reference implementation, it is important to keep in mind that later the code will be generated. Thus, the hand-written reference implementation must be a mechanical transformation of the model according to fixed algorithms. These algorithms can later be implemented to build the code generator.

A reference implementation needs not necessarily to include accessible application examples; they can later be implemented with the generator itself, which has the benefit of showing the usage of the generator, too. Rather, the reference implementation must include all features that are to be supported by the generator.

Thus, the state machines of the reference implementation look quite awkward and make no sense whatsoever, but they cover the important cases. They can be thought of as a test case reference for the code generator.

In the following sections I will present code snippets to explain the design of specific concepts. Figuring out what the best approach may be to implement a specific feature was one of the more challenging tasks of this thesis. After all, the code that is designed here will later be produced by the generator out of the users' models, and they will include that code in their application.

The full source code of the reference implementation can be found under `Sources/ReferenceImplementations/RI2/`. The folder `RI1/` contains a different approach for state machines that was discarded in favor for `RI2`, so it is not relevant here.

## 5.1 State Machine Classes

Before any state machine-specific model elements can be implemented in code, there must be a class definition that provides the container. This class must inherit from `QActive` and implement `init()` and `dispatch()`, as outlined in section *Quantum Platform Adaption Layer*.

The first part of the reference implementation uses a state machine called `AllTransitions`. Thus, the class definition looks as follows:

```
#pragma once
#include <qp_port.h>

class AllTransitions : public QActive {
    public:
        AllTransitions();
        virtual ~AllTransitions();

        virtual void dispatch(const QEvent * ev);
        virtual void init(const QEvent * ev = 0);
}
```

The following code snippets for the header file are all contained within the class definition.

## 5.2 Events and Signals

Having the class ready, it is possible to implement events and signals. The `AllTransitions` state machine contains the two signals `sigX` and `sigY`. According to QP, signals are implemented as enumeration, while parameterless events are implemented as immutable static member.

```
public:
    typedef enum Signals {
        sigX = Q_USER_SIG,
        sigY
    };

    static const QEvent evX;
    static const QEvent evY;
```

The applied naming conventions are obvious: Signal names start with “sig”, whereas event names start with “ev”. The `Q_USER_SIG` is required by QP to have the signal values at an offset. This is necessary because QP has some reserved internal signals, which use the enum values below `Q_USER_SIG`.

The association of signals and events takes place in the CPP file, where the events are initialized:

```
const QEvent AllTransitions::evX = {AllTransitions::sigX, 0};
const QEvent AllTransitions::evY = {AllTransitions::sigY, 0};
```

The `0` in the initializer denotes that the event is a static one and needs thus not be allocated from a memory pool. See [Samek08] for more information on event memory management.

## 5.3 State Hierarchy

Figure *RI State Machine “AllTransitions”* shows the state machine of the first reference implementation class. To map the state hierarchy to code, the *direct implementation* approach described in section *Evaluation Conclusion Example* will be adopted. It uses static member functions as state handlers and a function pointer to store the current state information.

Using static member functions is preferred over non-static member functions because non-static member function pointers are not the same size on all platforms and are known to perform badly [Samek08]. Also, it has the advantage that functions can be arranged in a hierarchy of nested inner classes to express the state hierarchy in a clear and concise way.

The following listing shows the state handler declarations for the `AllTransitions` state machine, followed by the current state pointer:

```
1  private:
2      typedef AllTransitions * Context;
3      typedef const QEvent * ConstEventPtr;
4
5      struct Sm {
6          struct S {
7              struct S1 {
8                  static void S11(Context context, ConstEventPtr ev);
9                  static void S12(Context context, ConstEventPtr ev);
10             };
11             struct S2 {
12                 static void S21(Context context, ConstEventPtr ev);
13                 static void S22(Context context, ConstEventPtr ev);
14             };
15         };
16     };
17
18     void (* currentState)(Context context, ConstEventPtr ev);
```

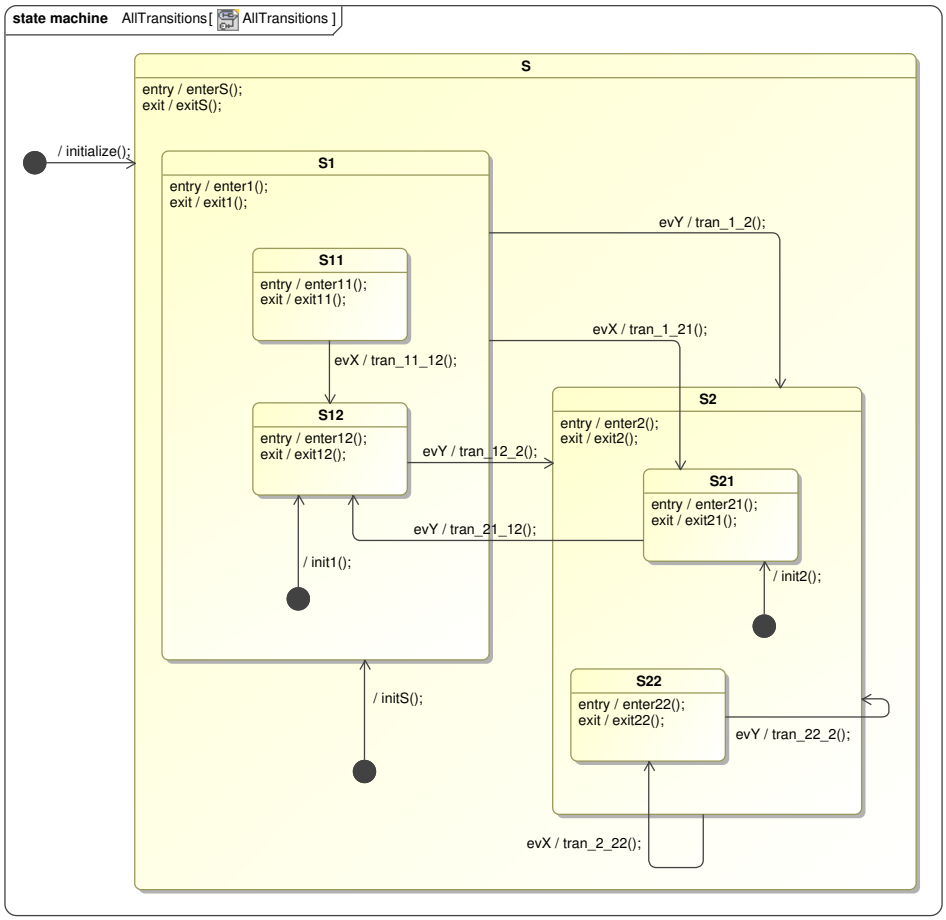


Figure 5.1: RI State Machine “AllTransitions”

The inner classes are implemented as `struct` instead of `class` just because the former have a public visibility of their members by default. The `Sm` forms the top-level element that is always present. It can be thought of as the state machine element of the metamodel (see also figure *UML State Machine*

*Metamodel*).

Apart from the visual encoding of the state hierarchy, organizing the state handler functions this way solves an important problem: States in different composite states may have the same name. If state handlers were implemented as a flat list of functions, there would arise a naming clash as soon as there are two different states with the same name.

Note that the nested `structs` do not have any impact on the runtime performance, they are a purely static construct.

## 5.4 State Handler Functions

Now that the state handler functions are declared in the header file, the next step is to implement them in the CPP file. The state handlers are responsible for executing the entry, exit and transition actions in the right order, and for setting the current state pointer accordingly.

All state handler have the same signature: They take a pointer to the enclosing object (like an explicit `this` pointer) and an immutable pointer to the event being processed. They return nothing. The two parameter types are defined by `typedefs`, see listing above.

The following listing shows the state handler for state `S12`:

```
1  void AllTransitions::Sm::S::S1::S12(Context context, ConstEventPtr ev) {
2      switch(ev->sig) {
3          case sigX:
4              context->exit_S_S1_S12(ev);
5              context->exit_S_S1(ev);
6              context->transition_S_S1_evX(ev);
7              context->enter_S_S2(ev);
8              context->enter_S_S2_S21(ev);
9              context->currentState = Sm::S::S2::S21;
10             break;
11         case sigY:
12             context->exit_S_S1_S12(ev);
13             context->exit_S_S1(ev);
14             context->transition_S_S1_S12_evY(ev);
15             context->enter_S_S2(ev);
16             context->initialTransition_S_S2();
17             context->enter_S_S2_S21(ev);
18             context->currentState = Sm::S::S2::S21;
19             break;
20     }
21 }
```

State handlers consist of a `switch` that distinguishes the event by its attached signal. A `case` block thus corresponds to one specific transition and contains the action sequence that is described by that transition.

The `S12` state handler illustrates nicely that it is not straightforward to find the action sequence for a transition. The algorithm for determining the action sequence, given a source state *S* and a target state *T*, is as follows:

1. Find the Least Common Ancestor (*LCA*) of *S* and *T*. The term Least Common Ancestor is defined by the UML and in my opinion not very clear, because it is not apparent what “least” means in the context of a state hierarchy.

It turns out that the *LCA* of *S* and *T* is the most specific (i.e. deeply nested) state that is an ancestor of both *S* and *T*. Note that *S* or *T* can be the *LCA* themselves in cases where *S* is an ancestor of *T*, or vice versa.

2. Leave all states until *LCA(S, T)*, starting at *S*, by executing the states' exit actions, if one exists. Do not leave the *LCA* itself.
3. Execute the transition action, if it exists.
4. Enter all states until *T*, starting at *LCA(S, T)*, by executing the states' entry actions, if one exists.
5. If *T* is a composite state, recursively follow the initial transitions in *T* until a simple state $T_{\text{init}}$ is reached. Execute all entry and initial transition actions along this path.

   If *T* is a simple state, then $T_{\text{init}} = T$.
6. Set the `currentState` variable to $T_{\text{init}}$.[1]

The transitions that are relevant for a state S are those that leave S directly, as well as all transitions inherited from ancestor states.

The state handler above, for example, inherits the transition with event `evX` going out of state `S1`. The transition with event `evY` of state `S1`, however, is masked by the transition that leaves `S12` directly, and is thus not inherited.

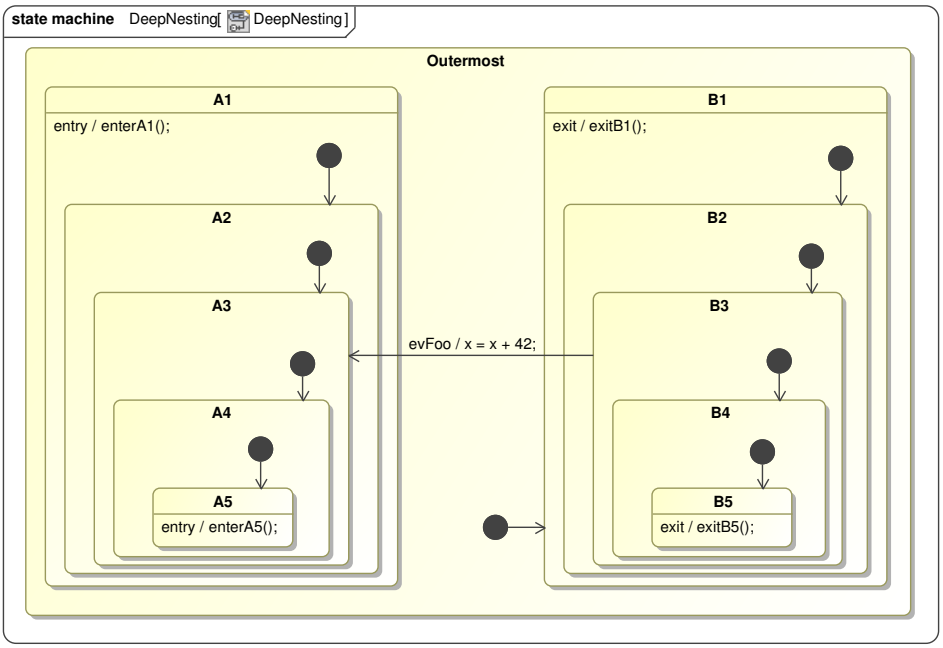


Figure 5.2: RI State Machine "DeepNesting"

Note that steps 5 and 6 of the action sequence algorithm above can also be used to determine the action sequence of the `init()` method. This works by interpreting the root state of the state machine as T.

Since the `AllTransitions` state machine contains only states of limited nesting level, another reference implementation has been created to verify that the generator works also with deeper hierarchies. Figure *RI State Machine "DeepNesting"* shows the state machine of the reference implementation that accomplishes this task.

[1] In fact the exact point in time when the `currentState` variable is reassigned does not matter, because during the transition the state machine is considered to be in an undefined state.

## 5.5 Action Wrappers

As the action sequences in the state handlers make clear, the actions are not called directly, but they are wrapped in a method. This is because actions may not only be specified as functions, but they could possibly contain arbitrary C++ code. The `DeepNesting` state machine contains an assignment to a member variable, for example.

Wrapping the actions in methods solves the following problems:

- The state handlers are static. Thus, parts of the actions that refer to non-static members lack context information. By implementing the action wrappers as member functions, this can be accounted for.
- Code duplication of action code may occur. With action wrappers, The user provided action code is located in the wrapper method only.

In order not to suffer from performance drawbacks, all action wrappers are declared `inline`.

## 5.6 Guards and Junctions

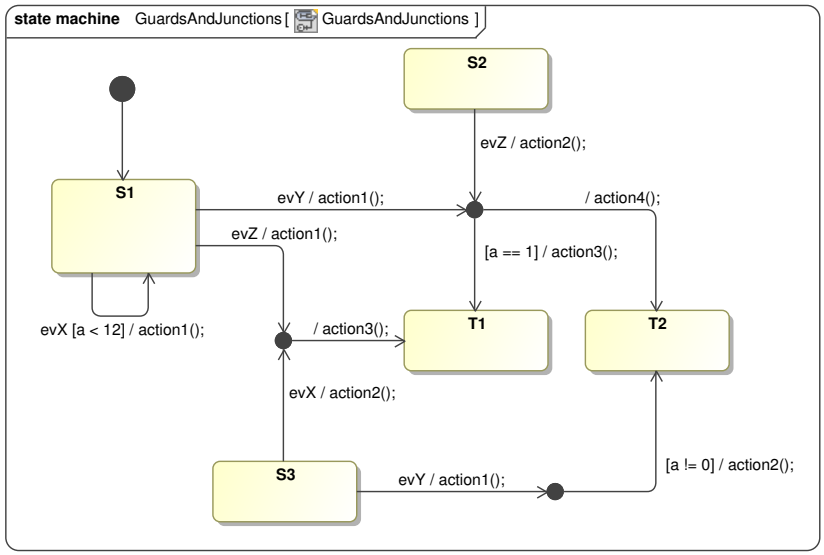


Figure 5.3: RI State Machine "GuardsAndJunctions"

Another state machine concept that is important for real world applications are guards. A guard is a condition that is attached to a transition, making the transition only fire if the condition is true. Guards can be simply added to a transition leading from a source state to a target state, but they may also be specified on a transition going out of a junction pseudostate. The latter opens up the possibility to create mutually exclusive transitions that are triggered by the same signal.

Junctions may also be used to bring two transitions together, for example to specify an otherwise duplicated transition action only once.

Figure *RI State Machine "GuardsAndJunctions"* illustrates the concept. The small black circles are junction pseudostates, guards are specified in square brackets.

The following listing shows the state handler for state `S1` in `GuardsAndJunctions`:

```
void GuardsAndJunctions::Sm::S1(Context context, ConstEventPtr ev) {
    switch(ev->sig) {
        case sigZ:
            context->transition_S1_sigZ(ev);
            context->transition_t3(ev);
            context->currentState = Sm::T1;
            break;
        case sigX:
            if(context->guard_t5(ev)) { // if a < 12
                context->transition_S1_sigX(ev);
                context->currentState = Sm::S1;
            }
            break;
        case sigY:
            if(context->guard_t2(ev)) { // if a == 1
                context->transition_S1_sigY(ev);
                context->transition_t2(ev);
                context->currentState = Sm::T1;
            }
            else {
                context->transition_S1_sigY(ev);
                context->transition_t1(ev);
                context->currentState = Sm::T2;
            }
            break;
    }
}
```

Simple guarded transitions are wrapped with an `if` statement, as the `sigX` block shows. Guards after a junction lead to `if` / `else` statements, as in the `sigY` block.

Note that the guard expressions need to be wrapped in member functions for the same reasons actions need to be wrapped.

## 5.7 Wrapper Method Naming

Unfortunately, finding a good naming strategy for transition action and guard wrappers is difficult. Taking the name of the transition source is not possible in all cases, because transitions may start at pseudostates, which do not have a name.[2]

In order to tackle the problem, the following definition is introduced:

[2] According to [OMG10], pseudostates may have a name, because they specialize *UML::Classes::Kernel::NamedElement* (through *Vertex*). In practice, however, they are normally left unnamed.

**Definition 2**

**Primary Transition** A transition that originates at a state. The state may be simple or composite, but since it is a state it has always a name.

**Non-primary transition** A transition that originates at a junction pseudostate and is thus preceded by another transition. In general, non-primary transitions cannot be assumed to end at a state, either.

Note that the UML does not distinguish these two types of transitions. However, it defines a name for a complete chain of transitions, leading from a source state to a target state: Such a set of transitions is called a *compound transition* [OMG10].

I define the naming strategy as follows:

| Wrapper | Naming |
|---|---|
| Primary transition actions | `transition_<source-state-fqn>_<signal-name>` |
| Non-primary transition actions | `transition_<transition-id>` |
| Initial transition actions | `initialize_<target-state-fqn>` |
| Entry actions | `enter_<state-fqn>` |
| Exit actions | `exit_<state-fqn>` |
| Guards | `guard_<transition-id>` |

In the table, `fqn` stands for fully qualified name, and in the context of a method name the state names of the ancestor path is separated by underscores (see listing below for examples).

A transition ID is a unique identifier for a transition. It is introduced specifically for the purpose of naming, it is not a construct defined by the UML. Basically, all transition-specific wrappers could be named by transition ID, but having good names when they are available increases readability of the generated code a lot.

Here are some examples of action wrapper declarations, taken from the `AllTransitions` and `GuardsAndJunctions` RI:

```
1  // Entry and exit actions (from AllTransitions)
2  inline void enter_S_S1_S12(ConstEventPtr ev);
3  inline void exit_S_S1_S12(ConstEventPtr ev);
4
5  // Initial transition action (from AllTransitions)
6  inline void initialize_S_S1_S12(ConstEventPtr ev);
7
8  // Primary transition actions (from AllTransitions)
9  inline void transition_S_S1_sigX(ConstEventPtr ev);
10 inline void transition_S_S1_sigY(ConstEventPtr ev);
11
12 // Non-primary transition actions (from GuardsAndJunctions)
13 inline void transition_t1(ConstEventPtr ev);
14 inline void transition_t2(ConstEventPtr ev);
15
16 // Guard (from GuardsAndJunctions)
17 inline bool guard_t2(ConstEventPtr ev);
```

## 5.8 Combining Generated Code with User Code

A common problem of code generators is that the generated code must be combined with the custom user code in some way. There are several solutions to this problem, some of which work only with certain target languages. In the following, the basic approaches are presented and the most suitable one is selected for this thesis.

### 5.8.1 Distributed Class Declaration

Maybe the straightforward way to combine user code with generated code is to place the two code fragments in different files. In C#, for example, a class declaration can be split up into multiple declarations in the form of *partial classes*. This feature is used by GUI editors extensively and would also work well for other code generators.

Unfortunately, partial classes are not available in C++. One could try to mimic the feature with `#include` statements and pull a class declaration together from different files:

```
1  // File example.h
2  class Example {
3      void userDefinedMethod1();
4      void userDefinedMethod2();
5
6      #include "gen/example.h"
7  };
8
9  // File gen/example.h
10 void generatedMethod1();
11 void generatedMethod2();
```

However, this approach does only work for very simple classes. As soon as the generated code itself needs some includes, they get included into the class scope, too, which is not valid C++ code anymore.

### 5.8.2 Combination by Inheritance

Another approach that fully separates user code and generated code is the use of inheritance: The user class simply inherits from the generated class. While this may be a clean solution in terms of code separation, it introduces unnecessary complexity, because there are now two classes where only one should be logically. Also, the access modifier of many fields and methods in the generated class needs to be loosened to `protected`, so that the user code can access them.

### 5.8.3 Protected Regions

Some generators integrate user code through *protected regions*. A protected region is a part of a source file that will never be touched by the generator. This is usually accomplished by placing special comments at the start and the end of the protected region:

```
class Example {

    void generatedMethod1();
    void generatedMethod2();

    // [##
    // start of user defined parts

    void userDefinedMethod1();
    void userDefinedMethod2();

    // end of user defined parts
    // ##]

};
```

In this example, everything between the `[##` and `##]` comments would not be modified by the generator. This introduces additional complexity in the generator, however, because the generator must copy the protected regions, then generate, and then copy the protected regions back into the newly generated files.

Another drawback of this approach is that the user may accidentally change some of the code outside the protected regions. When the generated code is in a different file, the chance of making this mistake is much smaller.

### 5.8.4 Separate Implementation

With C++, there is also a further possibility: Implementations of class members can be distributed arbitrarily over multiple source files. Each source file is then a separate compilation unit. Since the CPP files do not need to reference each other, this is a good technique to separate generated and user code of the implementation.

However, the header file that is included by the CPP files is still required to be one file only. Depending on the model elements that are generated, it may make sense to fully generate the header file. The consequence of this is that all header elements must be modeled, too, because the generator has otherwise no possibility to generate the right code.

### 5.8.5 Conclusion

Deciding on one of the above methods seems to be a consideration of drawbacks. In my opinion, the *Separate Implementation* approach is the one that is easiest to implement, and most importantly, also very clearly understandable for the user.

The generated files could also be set to read-only by the generator after the generation is finished. This truly eliminates the risk of accidentally editing generated files.

CHAPTER

SIX

# CODE GENERATOR

## 6.1 Generator Framework Components

The piece of software that acually transforms state machine models into C++ code is the code generator. It is based on the *Xpand* framework, which is part of the *Eclipse Modeling Project (EMP)*.

«The *Eclipse Modeling Project* focuses on the evolution and promotion of model-based development technologies (...) by providing a unified set of modeling frameworks, tooling, and standards implementations.» [WebEclipseModeling] As such, *EMP* is an umbrella project for many projects that are all devoted to modeling. The following parts of *EMP* are used in this thesis:

- *Xpand*, a statically-typed model to text (M2T) transformation language
- *UML2*, an Eclipse-based implementation of the UML meta model
- *Modeling Workflow Engine*, a framework to combine different tasks and run them sequentially
- *EMF*, the *Eclipse Modeling Framework*, is used by *Xpand* and *UML2*

The following subsections provide information on the different parts of the generator. The generator sources are contained in an Eclipse project called `ch.antiserum.scgen`, which can be found in folder `Sources/Generator/`. The file paths given in the sections below are all relative to the project directory.

### 6.1.1 Modeling Workflow

A workflow is basically nothing more than a sequence of some actions that need to be executed in order achieve a certain goal. Here, that goal is to produce C++ code.

Figure *Code Generator Workflow* shows the required tasks to get from model files to C++ code.

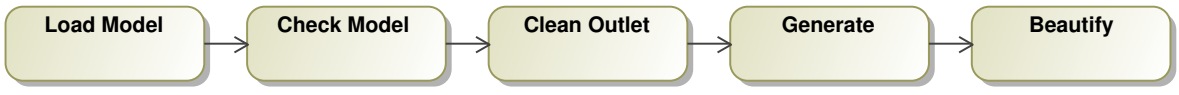


Figure 6.1: Code Generator Workflow

These tasks need to be specified in an XML document that forms the input to the *Modeling Workflow Engine (MWE)*. The following listing shows the *MWE* workflow used for the state machine code generator (some lines wrapped):

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <workflow>
3
4  <property file="workflowConfig/workflow.properties" />
5
6  <bean class="org.eclipse.emf.mwe.utils.StandaloneSetup">
7    <platformUri value=".."/>
8  </bean>
9
10 <!-- Load model -->
11 <component class="org.eclipse.emf.mwe.utils.Reader">
12   <uri value="${modelPath}" />
13   <modelSlot value="model" />
14 </component>
15
16 <!-- Check model -->
17 <component id="modelChecker" class="org.eclipse.xtend.check.CheckComponent">
18   <metaModel class="org.eclipse.xtend.typesystem.uml2.UML2MetaModel"/>
19   <checkFile value="checks/GeneratorRestrictions" />
20   <emfAllChildrenSlot value="model" />
21 </component>
22
23 <!-- Clean outlet directory -->
24 <component id="dirCleaner" class="org.eclipse.emf.mwe.utils.
25     DirectoryCleaner" directory="${outletPath}"/>
26
27 <!-- Generate -->
28 <component id="generator" class="org.eclipse.xpand2.Generator"
29     skipOnErrors="true">
30   <metaModel class="org.eclipse.xtend.typesystem.uml2.UML2MetaModel"/>
31   <expand value="${rootTemplate} FOR model"/>
32   <fileEncoding value="ISO-8859-1"/>
33   <outlet path="${outletPath}">
34     <postprocessor class="ch.antiserum.scgen.beautify.
35       UncrustifyBeautifier"/>
36   </outlet>
37 </component>
38
39 </workflow>
```

This file can be found under `src/workflow.mwe` and forms the entry point for the code generation.

The file as it is shown above is a rather typical one for an *Xpand* code generator. First, the properties file is loaded from `src/workflowConfig/`. Lines 11 to 14 load the serialized model file and store it internally. After the model checker verified that the model is sane, the directory cleaner erases all files in the outlet. This ensures that no dead files remain in the target directory. The component on lines 28 to 37 denotes the properties of the *Xpand* generator: UML is used as metamodel, the entry point of the generator is specified as `templates::PackageStructure::ModelRoot`, and the outlet path is retrieved from a property. Additionally, a beautifier postprocessor is specified for the outlet.

Workflows can be started in two different ways: Either the workflow is run directly from Eclipse as *MWE Workflow*, or the workflow file can be passed as argument to a `WorkflowRunner` that is started as ordinary Java program[1]. While the former is very useful when modifying the generator, the latter is the approach normally taken by the users of the generator. Refer to section *Tool Integration* for information on how the generator is started by its users.

[1] When the workflow is started through a `WorkflowRunner`, no running Eclipse instance is needed.

### 6.1.2 Xpand

The component in the workflow that actually generates code is *Xpand*. *Xpand* was developed as part of *openArchitectureWare (oAW)*, a platform for Model Driven Software Development. With the *Galileo* annual release of Eclipse, *oAW* migrated to the *Eclipse Modeling Project*.

Here, I give a very short introduction to *Xpand*, just enough to get started with the application of *Xpand* in this project. For further information, [Gronback09] provides an excellent introduction and reference, and [XpandDocs] contains the official reference documentation of *Xpand*.

While this is an introduction to *Xpand* itself, the concrete templates that generate state machine code are discussed in section *State Machine Xpand Templates*.

*Xpand* is a template-based code generation framework with a pluggable type system and polymorphic template invocation [XpandDocs]. It ships with an editor component for Eclipse that supports syntax highlighting, code completion and basic refactoring, and template debugging is possible.

An *Xpand* code generator is composed of a set of templates. Templates are stored in files with a `.xpt` extension and contain one or more definitions. A definition looks like this:

```
1 «DEFINE SimpleClass FOR uml::Class»
2     class «name» {
3         «EXPAND Operation FOREACH ownedOperation»
4         «EXPAND Attribute FOREACH ownedAttribute»
5     };
6 «ENDDEFINE»
```

This definition is named `SimpleClass` and is defined for model elements of type `uml::Class`. The output generated by this definition is a `class` keyword followed by the name of the passed `uml::Class` object, followed by an opening brace, a closing brace, and a semicolon. In between the braces, other definitions are called (or "expanded") for each `ownedOperation` and `ownedAttribute` element in the class. These elements are defined by the UML standard and implemented in *Eclipse UML2*, so they can be accessed from *Xpand* templates.

Note that definitions behave like functions. The example above has one argument (the `uml::Class` instance) and "returns" the generated C++ code, there are no side effects.

### 6.1.3 Xtend

Expressions for selecting model elements, like the `ownedOperation` and `ownedAttribute` above, can become complicated quickly. Consider the following expression that selects all simple states contained in a given state `s`:

```
1 s.region.subvertex.typeSelect(uml::State).select(e | e.isSimple())
```

Expressions like these are necessary to traverse the model in the right way, but clutter the *Xpand* templates heavily. In order to improve the readability, such expressions can be swapped out to extension files (.ext). These files are populated with functions written in a simple functional language called *Xtend*, which is also part of the *Xpand* generator framework.

The expression above could be simplified by using *Xtend* functions like this:

```
1 s.simpleSubstates()
```

The function `simpleSubstates` must then be implemented in an extension:

```
simpleSubstates(uml::State this):
        region.substates().select(e | e.isSimple());

substates(uml::Region this):
        subvertex.typeSelect(uml::State);
```

The idea is to divide complex expressions repeatedly until they become small functions with obvious meaning. This functional composition improves the readability of the templates a lot, which is crucial if later the templates are to be adapted to specific project requirements. Thus, this technique is used extensively in this thesis.

### 6.1.4 Check

One of the big advantages of practising model driven development is that the model can be checked before generating code. This means that architectural rules and project-specific guidelines can be enforced on an abstract level. This kind of checks are normally not possible with code-only development, because the abstract information of the model is hidden in the complexity of the code.

Checking for cyclic dependencies, for example, is easy within a model checker, but difficult to establish on source files.

The *Check* component is part of *Xpand* and facilitates model checking. *Check* rules are specified in a `.chk` file that can be used in the workflow to validate the model before generating any code (recall the `modelChecker` component in the *Modeling Workflow*).

Model checks are normally used for project-specific rule enforcement. For this thesis, however, it is also useful, because it enables the validation of applied generator restrictions. The generator does not support parallel regions, e.g., which is a constraint that is more restrictive than standard UML. It is thus reasonable to check the model for constructs that are not supported by the generator.

The following listing shows and excerpt of the check file that enforces the generator restrictions. The complete file can be found under `src/checks/GeneratorRestrictions.chk`.

```
context uml::State ERROR
  "There may be only one region in a state (no parallel regions)." :
  region.size <= 1;

context uml::Transition ERROR
  "There may be only one trigger assigned to each transition." :
  trigger.size <= 1;

context uml::Pseudostate if kind == uml::PseudostateKind::initial ERROR
  "Initial pseudostates must have exactly one outgoing transition." :
  outgoing.size == 1;
```

This shows the simple and straightforward syntax of check files: The context, to which the rule is to be applied, is followed by the message and the rule itself. The keyword `ERROR` specifies that the check fails if the rule is violated. Alternatively, the keyword `WARNING` prints only a message but the workflow continues with code generation anyway.

### 6.1.5 Source Code Beautification

As can be seen in the workflow specification in section *Modeling Workflow*, there is a beautifier attached to the outlet. This means that every generated C++ file is processed by the beautifier after generation.

Beautifier are important for template based code generation, because they enable the generator developer to indent the templates according to the template language instead of the target language. Hence, the main reason for having a beautifier is to correct the indentation of the generated files.

*Xpand* lacks a built-in C++ beautifier, but there are plenty of good open source beautifiers for C++ around. One of the more flexible beautifier for C++ is *Uncrustify*. It is highly configurable, which is good for this thesis, because the configuration can later be adapted if that is required by the project coding guidelines.

*Uncrustify* is typically called from the command line with the language and the configuration file as parameter, as well as the file to be beautified. The result is normally printed to STDOUT, but *Uncrustify* can be configured to directly modify the source file. The following command is used to beautify the generator output:

```
1 uncrustify -c <config> -l CPP --replace --no-backup <source-file>
```

Of course, for this to work the *Uncrustify* executable must be on the path. The configuration file can be found in the `config` folder of the beautifier *Eclipse* project.

Having the beautifier ready, the only thing that remains is an adapter class, so that *Uncrustify* can be called from the workflow. This adapter is implemented in a separate Eclipse project called `ch.antiserum.scgen.beautify`, which is located alongside the generator project. In order to be callable from the workflow, the `UncrustifyBeautifier` adapter implements the `org.eclipse.xpand2.output.PostProcessor` interface. This enables the beautifier to be called as outlined in *Modeling Workflow*.

The adapter class itself is very simple. The method `afterClose()`, implemented according to the `PostProcessor` interface, is called by the workflow engine after the code has been generated and written to the file. In this method, *Uncrustify* is spawned as a `java.lang.Process` and used to beautify the just generated code with the command line options shown above.

Also, the informational messages that *Uncrustify* prints to STDERR are captured and redirected to the *Log4j* facility.

## 6.2 State Machine Generator Implementation

The code generator is, as already mentioned, implemented as a set of *Xpand* templates and *Xtend* files. This section highlights the central design decisions, template organisation, applied idioms and some *Xpand* definitions.

### 6.2.1 Template Organization

In order to be easily extensible and maintainable, the generator templates need to be structured in a sensible way. To achieve this, template files can be split up and stored in different packages. These packages are the same as Java packages and can be arbitrarily arranged in a hierarchical sense.

The following tree shows the packages in the `src` directory of the generator *Eclipse* project:

```
src
|-- checks
|   `-- GeneratorRestrictions.chk
|-- extensions
|   |-- classes
|   |   |-- Dependencies.ext
```

```
|   |   |-- Operations.ext
|   |   `-- Properties.ext
|   `-- stateMachines
|       |-- _All.ext
|       |-- GuardedTransitions.ext
|       |-- GuardExtensions.java
|       |-- Guards.ext
|       |-- InitialStates.ext
|       |-- SignalExtensions.java
|       |-- Signals.ext
|       |-- StateHierarchy.ext
|       |-- StateMachines.ext
|       |-- StateNaming.ext
|       |-- TransitionExtensions.java
|       |-- Transitions.ext
|       `-- WrapperMethods.ext
|-- templates
|   |-- stateMachineClasses
|   |   |-- header
|   |   |   |-- ActionMethods.xpt
|   |   |   |-- Class.xpt
|   |   |   `-- StateMachines.xpt
|   |   |-- implementation
|   |   |   |-- ActionMethods.xpt
|   |   |   |-- Class.xpt
|   |   |   |-- StateMachines.xpt
|   |   |   `-- Transitions.xpt
|   |   `-- Main.xpt
|   `-- PackageStructure.xpt
|-- workflowConfig
|   `-- workflow.properties
`-- workflow.mwe
```

The `checks` package contains model checker files (`.chk`).

In the package `extensions` reside the *Xtend* files (`.ext`) and the extensions that are implemented in Java (`.java`). They can be used from *Check* files and from *Xpand* templates, so they form a kind of shared resource. Extensions are divided into class-specific extensions and state machine-specific extensions. The special extension `stateMachines/_All.ext` is an aggregator of the others, so that templates that need almost all state machine extensions need only to import `_All.ext`. The extension files group functions with a similar purpose or scope.

The `templates` package contains the *Xpand* templates (`.xpt`), where the `PackageStructure` template forms the entry point from the workflow. Templates for state machine classes are divided into `header` and `implementation`, where the former generate `.h` and the latter `.cpp` files.

### 6.2.2 Xpand/Xtend Idioms

When working with a certain language, some patterns soon provide themselves useful for specific tasks and are used often. These language-specific patterns are commonly called idioms.

In this section I describe the idioms that I identified while working on the generator. I present them here because having understood these idioms simplifies understanding the whole code generator a lot.

### Extension-based Model Traversing

#### Intent

Structure template code in a clear and maintainable way. Separate code generating parts from model traversing parts.

#### Problem

Traversing the model happens by recursively selecting sub elements of a model element. The expression for selecting those elements can basically reside either in the *Xpand* templates or in *Xtend* extensions. Distributing them without a clear aim makes it difficult to find code later on.

#### Solution

Consequently put code for model element selection in extensions, and use template definitions only for code generation and definition grouping.

### Implicit Null Checking

#### Intent

Increase code readability by reducing the amount of code and performing implicit checks for optional elements.

#### Problem

Model elements typically have child elements that are optional. For example, states may or may not have an entry action that is executed on entering the state. A template definition for entry action methods could thus look as follows:

```
«DEFINE EntryAction FOR uml::State»
    «IF entry != null»
        inline void «s.entryActionMethodName()»(ConstEventPtr ev);
    «ENDIF»
«ENDDEFINE»
```

The calling code, assuming we are in a `DEFINE` for a `uml::State`, would be:

```
«EXPAND EntryAction FOR this»
```

This approach generates code as expected, but leads to many `IF` statements that clutter the templates.

#### Solution

Use Implicit Null Checking as implemented in *Xpand* to only expand a definition for arguments that are not null. This requires that the element, which needs to be checked for null, is the primary argument to

the definition (the one after the FOR keyword). The example above could be rewritten with Implicit Null Checking as follows:

```
«DEFINE EntryAction(uml::State s) FOR uml::Behavior»
    inline void «s.entryActionMethodName()»(ConstEventPtr ev);
«ENDDEFINE»
```

with calling code

```
«EXPAND EntryAction(this) FOR entry»
```

Since DEFINE's are only expanded if the element they are specified for is not null, this is equivalent to the explicit IF version above. Note that the uml::State is required in the DEFINE, thus it is passed as additional argument.

A slightly different, but conceptually similar version of this idiom occurs with lists: When an «EXPAND t FOREACH xs» construct is used, the template t is not expanded if the list xs is empty.

#### Well-formed Model

#### Intent

Improve readability by not cluttering template and extension code with error checking.

#### Problem

The generator must take certain assumptions on the model structure, because the generator would otherwise become very general and must handle all corner cases. Controlling if the assumptions are met leads to code bloat and obscures the true intention of a definition or extension.

#### Solution

Use *Check* to validate the model before generating code. All assumptions that the generator makes must be enforced by applying respective model check rules. Since the model checker always runs strictly before the generator, and no generation is performed at all when the checker fails, the generator can safely assume a well-formed model.

### 6.2.3 State Machine Xpand Templates

In this section I will highlight some of the *Xpand* templates that generate state machine code. A good reference when looking at the templates is the *Reference Implementation*, because it dictates how the code must look after generation.

Two naming conventions are important to understand the templates: Extensions that start with all, as in allSimpleStates(), return all described elements that are descendants of the argument the function is applied to. In contrast, extensions without the all, such as simpleStates(), return only the immediate children, but no elements that are further down the hierarchy.

The other naming convention is the use of only at the end of extension names. These functions return the given argument if it matches the described condition, or null otherwise. For example, the func-

tion `simpleStateOnly()` takes a state and returns the same state if it is simple, but `null` if it is composite.

The generator templates that are described in the following can be found in `Sources/Generator/ch.antiserum.scgen/src/templates/`.

### Header and CPP Files

One question that comes up when implementing the templates is where the distinction between header and CPP files should be made. I make this distinction early in the template hierarchy, because it clearly separates templates for header and implementation. The following listing shows the template definition that splits header and implementation templates:

```
1  «DEFINE StateMachineClass FOR uml::Class»
2
3    «FILE name.toString() + ".h"»
4      «EXPAND header::Class::StateMachineClass»
5    «ENDFILE»
6
7    «FILE name.toString() + "_sm.cpp"»
8      «EXPAND implementation::Class::StateMachineClass»
9    «ENDFILE»
10
11 «ENDDEFINE»
```

As can be seen, the following templates are split into a `header` package and an `implementation` package.

### Class Declaration

The class declaration templates generate the includes, the required forward declarations, and the class code itself. Within the `class` block, all templates that generate their code into the class are called:

```
1  «DEFINE StateMachineClass FOR uml::Class»
2    #pragma once
3
4
5    #include <qp_port.h>
6
7    «EXPAND ForwardDeclaration FOREACH dependencySuppliers()»
8
9    class «name» : public QActive {
10
11     «EXPAND CtorDtor»
12     «EXPAND Operations»
13     «EXPAND Properties»
14     «EXPAND StateMachines::StateMachine FOR stateMachine()»
15   };
16 «ENDDEFINE»
```

Note that the operations and properties are only declared by the generated code. The operations are expected to be implemented in user code. See *Combining Generated Code with User Code* for more information on this.

### State Hierarchy

The last EXPAND statement in the previous listing expands the state machine declarations into the class body. The following listing shows that template:

```
«DEFINE StateMachine FOR uml::StateMachine -»
  public:
    typedef enum Signals {
      «EXPAND SignalDefinition(signals()) FOREACH signals() -»
    };

    // Static events
    «EXPAND EventDefinition FOREACH signalsWithoutAttributes()»

    «EXPAND DispatchMethod -»
    «EXPAND InitMethod -»

  private:
    typedef «context.name» * Context;
    typedef const QEvent * ConstEventPtr;


    // State and initial transition action wrappers
    «EXPAND ActionMethods::ActionMethodDefinitions FOREACH
      region.allStates()»

    // Primary transition action wrappers
    «EXPAND ActionMethods::TransitionAction FOREACH
      region.allPrimaryTransitionsWithEffect()»

    // Non-primary transition action wrappers (arbitrary names)
    «EXPAND ActionMethods::TransitionAction FOREACH
      region.allNonprimaryTransitionsWithEffect()»

    // Guard wrappers (arbitrary names)
    «EXPAND GuardMethodDefinition FOREACH
      region.allGuardedTransitions()»

    // The state hierarchy with the state handler functions

    «EXPAND StateHierarchy FOREACH region»

    «EXPAND CurrentStatePointerDefinition»
«ENDDEFINE»
```

The interesting parts here are the EXPAND statements for the different types of wrapper methods and the EXPAND for the StateHierarchy. The latter is shown in the following listing:

```
«DEFINE StateHierarchy FOR uml::Region -»
  struct «nameInHierarchy()» {
    «EXPAND StateHandlerDefinition FOREACH simpleSubstates()»
    «EXPAND StateHierarchy FOREACH compositeSubstateRegions() -»
  };
«ENDDEFINE»
```

This is the first non-trivial template. It produces a struct and expands two other templates in the body of the struct: The first generates code for the state handlers, which is trivial, and the other is the StateHierarchy template again; it recursively expands itself.

Note that this recursion always terminates, because well-formed models contain a finite state hierarchy and thus the function `compositeSubstateRegions()` eventually returns an empty list, resulting in no further expansions. This is an application of the *Implicit Null Checking Idiom*.

Since each `StateHierarchy` expansion produces a new `struct`, the generated code ends up with nested `structs` that correspond to the state hierarchy of the model (compare *Reference Implementation – State Hierarchy*).

### Class Implementation

The starting point for templates that generate into the CPP files is a template that encapsulates the class implementation of a state machine class, analogous to the class declaration in the header templates:

```
1  «DEFINE StateMachine FOR uml::StateMachine»
2    «EXPAND EventDefinition(this) FOREACH signalsWithoutAttributes() »
3
4    «EXPAND DispatchMethod»
5    «EXPAND InitMethod»
6
7    «EXPAND ActionMethods::ActionMethodImplementations FOREACH
8      region.allStates()»
9    «EXPAND ActionMethods::TransitionActionImplementation FOREACH
10     region.allPrimaryTransitionsWithEffect()»
11   «EXPAND ActionMethods::TransitionActionImplementation FOREACH
12     region.allNonprimaryTransitionsWithEffect()»
13   «EXPAND GuardMethod(context) FOREACH
14     region.allGuardedTransitions()»
15   «EXPAND SimpleState(context) FOREACH
16     region.allSimpleStates()»
17 «ENDDEFINE»
```

Again, there are `EXPAND` statements for the different wrapper methods, followed by the expansion of the `SimpleState` template. Note that composite states do not require code in the CPP file, because the state hierarchy is flattened and the composite states manifest in the generated code only as the hierarchy of `structs` in the header file.

The `SimpleState` template looks as follows:

```
1  «DEFINE SimpleState(uml::BehavioredClassifier context) FOR uml::State»
2    void «context.name»::«stateFqn()»(Context context, ConstEventPtr ev) {
3      «EXPAND EventSwitch FOR stateWithRelevantOutgoingTransitionsOnly() -»
4    }
5  «ENDDEFINE»
```

The function header and body is generated, and in the body the `EventSwitch` template is expanded. Note that `stateWithRelevantOutgoingTransitionsOnly()` returns the `uml::State` of the `SimpleState` template if it has relevant outgoing transitions, or `null` otherwise.

States without relevant outgoing transitions cannot be left and thus require no `switch` for the events. This, again, is the *Implicit Null Checking Idiom* in action.

### Action Sequences

`EventSwitch` and the next few templates in the chain are trivial, they generate the `switch` statement and the `case` labels. The next template that is interesting is `ActionSequence`, which implements

the algorithm described in *Reference Implementation – State Handler Functions*. It is shown in the following listing:

```
1  «DEFINE ActionSequence(uml::State affectedSimpleState) FOR
2      List[uml::Transition] -»
3    «EXPAND LeaveStateUntil(leastCommonAncestor(mainSource(), mainTarget()))
4      FOR affectedSimpleState -»
5    «EXPAND TransitionAction
6      FOREACH this -»
7    «EXPAND EnterStateFrom(leastCommonAncestor(mainSource(), mainTarget()))
8      FOR mainTarget() -»
9    «EXPAND Initialize FOREACH mainTarget().region -»
10 «ENDDEFINE»
```

Here, things start to get more complicated, mainly because an action sequence must be generated for a compound transition. This is why the template is defined for a list of transitions.

If there had been an explicit metaclass for compound transitions, this part of the templates would have been a lot simpler. Unfortunately, such a metaclass does not exist, thus compound transitions need to be expressed as lists of transitions.

Line 3 expands the `LeaveStateUntil` template, which generates a call to the exit action wrapper and recursively expands itself until the LCA is reached. The functions `mainSource()` and `mainTarget()` take a compound transition and return the state at the beginning or the end, respectively.

Line 5 expands a trivial template that generates the call to the transition action wrapper.

Line 7 expands `EnterStateFrom`, which is similar to `LeaveStateUntil`, but operates in the other direction, that is from the LCA to the `mainTarget()`.

Finally, line 9 recursively follows the initial transitions of the `mainTarget()` state, if it is composite.

### 6.2.4 Applied Metamodel Restrictions

This chapter made it clear: There are many UML state machine features that are not realized in the generator templates.

Many UML features are not required in practice, and some others are just syntactical sugar for UML models. Consequently, implementing all of them makes no sense. This section discusses the decisions that lead to the set of features that are now implemented, given the limited time for the thesis.

Restricting the UML metamodel has also two important advantages: The first is that the implementation of the generator becomes simpler, which is important because the generator will be modified for specific project requirements. If the generator is very complicated, developers will refrain from making any changes at all, because they fear that they may break something.

The second advantage stems from the fact that the state machines form a kind of a DSL themselves. Restricting that DSL makes it effectively easier to learn, which is a good thing [Fowler10].

#### Parallel Regions

The most obvious feature that is not implemented is support for parallel regions. The main reason for this is the considerable complexity that parallel regions introduce. QP does not support parallel regions either, which means that they would have to be implemented on top of QP.

However, leaving them away is not much of a problem, because they can be simulated by forwarding events to multiple secondary state machines. These secondary state machines then act as parallel regions. [Samek08] contains an in-depth discussion of this approach, which is also the solution when using QP without a generator.

Without having parallel regions, the *Fork* and *Join* pseudostates become superfluous, too. Consequently, they have been left away.

### Pseudostates

Many of the other pseudostates are not required either: The *History* pseudostates can be implemented manually, *Choice* can be replaced by *Junction* with a rearrangement of the guard expressions. *EntryPoint* and *ExitPoint* are a purely graphical optimization and introduce no semantics whatsoever.

Thus, only the *Junction* and *Initial* pseudostates are implemented for this project. Others may be added at a later time, since extending the current feature set of the generator is not much of a problem.

### Multiplicity Restrictions

With the UML being very general, there are often "to-many" relations where just a "to-one" relation is actually required. One example of this is the relation between the *Transition* and the *Trigger* metaclasses (see figure *UML State Machine Metamodel*). This relation states that a transition may have an arbitrary number of triggers, but in practice a transition has either one trigger or none.

Thus, the implementation allows one trigger at most. There are other relations where similar reasoning applies.

### Restriction Enforcement

For the users of the generator it is obviously not nice if they use a UML feature and the generator produces some broken code, just because that feature is not implemented. It is thus important that the model checker, which runs before the generator, verifies that the model does not use any of the unsupported features.

If the checker finds a feature that is not supported, it produces a clear error message so that the user knows which part of the model needs to be changed.

Section *Generator Components - Check* contains more information on the model checker.

CHAPTER

SEVEN

# TOOL INTEGRATION

The developers that work with the code generator will often have to switch from the UML editor to the IDE and back, generating code every time they make a change in the model. It is thus important to have as tight an integration between the UML editor and the code generator as possible.

## 7.1 MagicDraw

The UML editor of choice for this thesis is *MagicDraw* from *NoMagic, Inc.* This is because *MagicDraw* can export *Eclipse UML* files directly[1]. Hence, the goal is to be able to call the *MWE* workflow from within *MagicDraw*.

### 7.1.1 MDA Integrations

*MagicDraw* offers an integration component for *Xpand* / *MWE*[2] as part of the *MDA Integrations*, which are available from *MagicDraw Standard Edition* upwards. This integration provides an *MWE* component to export a *MagicDraw* project to *Eclipse UML*.

Unfortunately, the component starts up a full *MagicDraw* instance to export the model, even when there is already an instance running and the project loaded [NoMagic10]. This takes about 30 to 40 seconds on a recent PC. That is clearly too long to be usable in model development cycles. Thus, the *MDA Integration* cannot be used here.

One application where it could be useful nonetheless, is in automated builds in a continuous integration setup.

### 7.1.2 External Tools Integration

Another possibility in *MagicDraw* is to start external tools from the tool bar. The tools can be configured in the environment settings of *MagicDraw* and can thereafter be started by selecting the tool from a drop down on the "external tools" tool bar and hitting run. By using the appropriate settings, *MagicDraw* exports the model to *Eclipse UML* files before running the external tool.

With this facility we have thus the possibility to start the *MWE* workflow by using the command line interface to invoke a `WorkflowRunner` (see *Modeling Workflow*). However, in order to enable a more flexible integration, some glue scripts are necessary.

[1] Since the XMI files are standardized, however, it should be possible to use other tools with no or only minor changes to the generator.

[2] In the *MagicDraw* docs the old name *openArchitectureWare (oAW)* is still used instead of *Xpand* or *MWE*.

## 7.2 Integration Scripts

Even though they are used in conjunction with *MagicDraw*, these scripts can easily be adapted to whatever UML editor wanted. They are written as Windows batch scripts. This is neither a very readable nor comfortable scripting language, but it is available on any Windows platform and thus we do not introduce further dependencies like a Python interpreter, e.g.

The following scripts are called in order to invoke the *MWE workflow*:

1. Redirection script. It changes to the `scripts` directory in the working directory of the opened *MagicDraw* project, which is passed in as parameter. This is necessary because *MagicDraw* is unable to start external tools in the working directory of the project.
2. Project configuration script. This script is stored per project. The paths to the model files and to the outlet are configured in this script. It then calls the model converter script and starts the workflow runner script in a new process. The new process has the effect of opening a new window, which is required to see the workflow output. *MagicDraw* is unable to redirect STDOUT of external tools to the message window, thus this is a necessary workaround.
3. Model converter script. Since *MagicDraw* exports as *Eclipse UML2* version 2.0.0, but *Eclipse Helios* works with *Eclipse UML2* version 3.0.0, a *sed* script is required to change the namespace in the XMI file accordingly. The script does **not** change the model, however.
4. Workflow runner script. This script configures the Java class path and starts the `WorkflowRunner` Java program. It passes in the *MWE* parameters as received from the project configuration script.

Of course it would also be possible to call script 3 and 4 directly from *MagicDraw*. But then the generator would be bound to one specific project, which is very inflexible.

The following table shows the locations where the respective scripts can be found:

| Script | Location |
|---|---|
| 1 | `Sources/Integration/MagicDraw/generateProject.bat` |
| 2 | `Sources/Examples/CandyMachine/Model/scripts/startWorkflow.bat` |
| 3 | `Sources/Integration/MagicDraw/convertModelFiles.bat` |
| 4 | `Sources/Generator/ch.antiserum.scgen/scripts/runWorkflow.bat` |

The scripts try and use relative paths as much as possible, but sometimes absolute paths are unavoidable. In those cases, the absolute path is made explicit by specifying it as variable inside the script.

CHAPTER

EIGHT

# CONCLUSION

## 8.1 Project Achievements

After an analysis of various existing state machine frameworks, QP was selected to be used as generator target environment. With QP implementing all the necessary low-level constructs to build state machines, I have been able to fully concentrate on the modeling and generator aspects. The UML is surprisingly complex when it comes to details, and those complexities can also be sensed in the generator templates.

But, the generator is in my opinion well-manageable and extensible. It can be adapted to specific project requirements with low effort. After all, this was one of the goals of this thesis, because the generator will unlikely be used without project-specific extensions. Even the porting to a different execution engine should be possible with reasonable effort.

The *Xpand* framework is relatively easy to learn and provides all the nuts and bolts that are required to load and process a model, perform checks and generate code. Maybe the biggest hurdle is to get accustomed to the functional style programming that is required within *Xpand*.

It became clear quite fast that *Xpand* is normally used to generate Java code; there is no built-in beautifier for C++, for example. But, with some small extensions to the generator infrastructure, *Xpand* became very usable in non-Java environments, too.

The implemented generator templates produce fast, readable code for the state machines. In fact, the resulting code is asymptotically faster than the original QP event processor. This is due to the fact that the generator can pre-compute some algorithms that standalone-QP must perform at runtime.

Apart from the UML editor, all the tools and frameworks that are used here are open source[1]. This shows that a pluggable, open generator solution is possible. I think I have been able to prove that solutions like this could constitute a real alternative to expensive MDSD tools.

Clearly, this generator implements only a subset of what mature MDSD tools provide. But I feel that the open generator templates provide a profound conceptual advantage over the closed generators of commercial tools. Thus, I regard the open approach to be able to outmatch the closed tools in practice, given that the open generators are well-designed an cover a certain minimum of features.

[1] There are open source UML editors, but most of them are clumsy and offer a very limited feature set. One of the more promising editors in conjuction with *Eclipse* modeling is *TopCased*, which is able to directly store the model as *Eclipse UML* files.

## 8.2 Practical Considerations

The generator in its current form is very well ready to be used in real world projects. There are some points to keep in mind, however. This section outlines some of the drawbacks of the generator and proposes workarounds.

### 8.2.1 Parametrized Events

One of the biggest shortcomings that I see for real applications is the lack of support for arbitrarily parametrized events. Such events are supported by QP, where they are called "dynamic events", but the generator is currently unable to produce the required event class declaration.

It is still possible to use parametrized events, however. The generator does **not** generate static event declarations as soon as there are event parameters, which would be wrong. Thus the developer has the possibility to declare the dynamic event manually. The signal for the dynamic event can be pulled from the ordinary signal enumeration. In the action methods, the parameter can then be accessed by down-casting the event to the specialized event class to access the parameters.

### 8.2.2 Structural Models

The focus of this thesis lay on the state machines, which are *Behavioral Models*. Hence, the support for generation of other models is limited to the necessary minimum.

A class relationship, for example, is a part of a *Structural Model*. Structural modeling elements like packages, dependencies, and relations of all kind are ignored by the current generator implementation. Thus, only the state machine-specific part of a model can be used to generate code.

### 8.2.3 Types and Type References

Also, types that are referenced in operations and attributes are taken as they are, and only their name is of relevance. If, for example, an operation takes a parameter of type `ExampleClass`, the expected code is normally a pointer to `ExampleClass`. But the generator just prints the name of the parameter type as-is, thus generating a call by value.

To get a pointer nonetheless, a new type with name `ExampleClass *` needs to be defined in the model and referenced in the operation. That setup produces the expected code.

### 8.2.4 UML Editor

The editor of choice for this thesis was *Magic Draw*, primarily because it is able to export *Eclipse UML* files directly. Other editors can be used with this generator as well, given it is somehow possible to produce model files that can be loaded by *Eclipse*.

Once the model files can be loaded, the generator output using the reference models can be compared to the reference implementation. This verifies that the generator works for the new editor. If not, some model elements are used differently, and the generator must be changed accordingly. These cases should be rare, however, because the generator is implemented according to [OMG10], and not according to any editor-specific model structures.

## 8.3 Outlook and Improvement Opportunities

Section *Practical Considerations* gives already a hint for possible improvements. Especially the parametrized events would be a feature that makes the generated code more concise, because the events would not have to be defined manually.

Also, better support for generation of types, interfaces, classes, relations and dependencies would be nice to have. With a well-implemented generator for structural models, a model checker for whole software systems could be built. This would extend the benefits of having a generator beyond the scope of state machines.

The generator is targeted at event driven software systems. These often follow the same patterns for object interaction, network communication and other common tasks. These tasks could be assisted with a set of modeling extensions in the form of stereotypes. Stereotyped elements would then trigger other generators when generating code. With the appropriate libraries in the background, this can be thought of as a model-based DSL for event driven systems.

Another beneficial extension that I see would be a unit testing environment for event driven systems. Testing reactive classes is often cumbersome with traditional unit testing frameworks. Extending such a framework for reactive classes would be a nice add-on to the current generator solution.

APPENDIX

A

# INCLUDED EXAMPLES

This section shortly describes the intent of the examples provided with the generator. They can be found in the directory `Sources/Examples/`.

## A.1 Simple Lamp

This is a very simple example consisting of a two-state state machine that models a lamp that can be turned on or off. Figure *Lamp Example State Machine* shows the state machine of the lamp.

It was implemented twice, once as standalone-QP application and once with use of the generator. The two versions can be found in folder `QpLamp/` and `GenLamp/`, respectvely.

Having both versions available is a straightforward way to compare the basic building blocks of the two approaches.

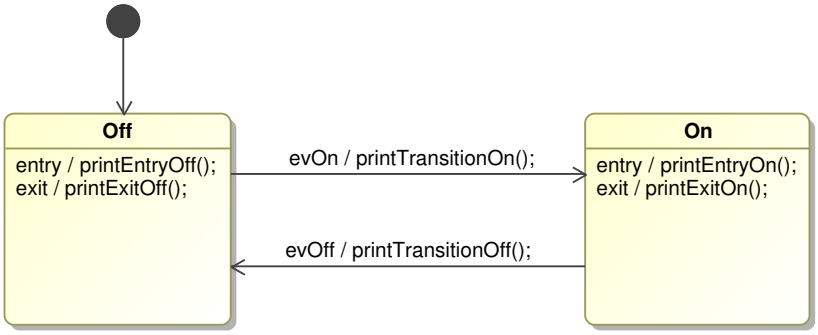


Figure A.1: Lamp Example State Machine

## A.2 Candy Machine

The example in folder `CandyMachine/` is a bit more complex than the lamp example. It is implemented with the generator and features a main state machine, which models a candy machine, and a helper state machine to model an indicator lamp. The indicator lamp has been introduced to show the interaction of two state machines. The idea is that as soon as enough coins have been inserted, the indicator lamp lights up and signals the user that the candy is ready.

The following figures illustrate the example:

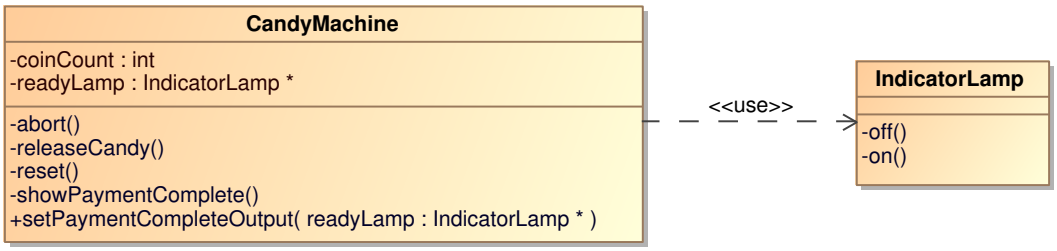


Figure A.2: Candy Machine Classes

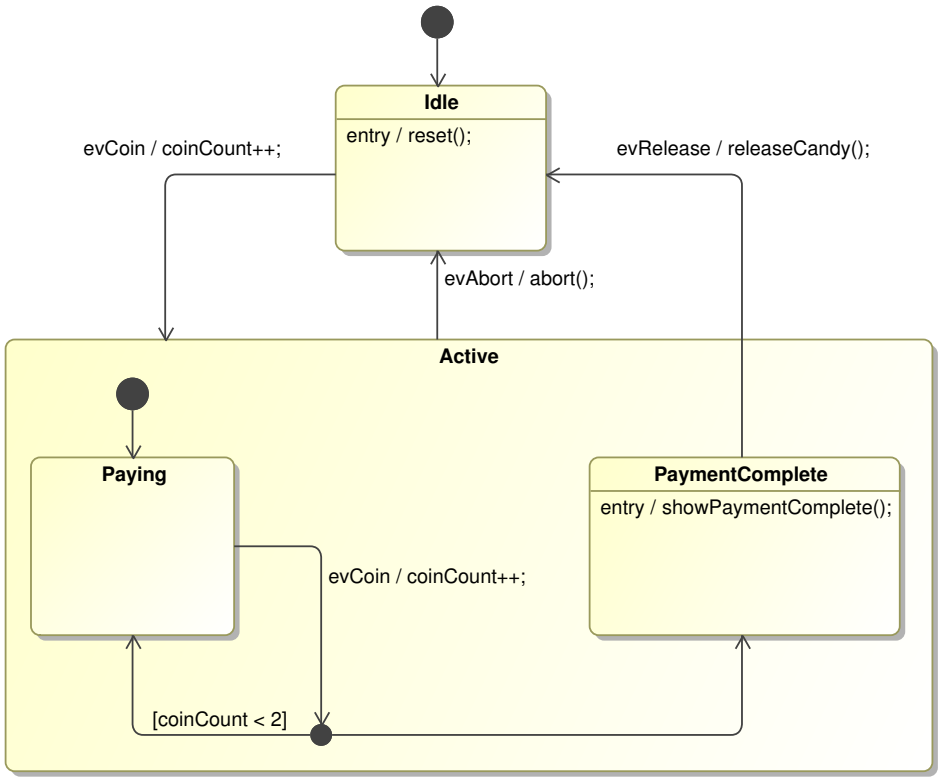


Figure A.3: Candy Machine State Machine

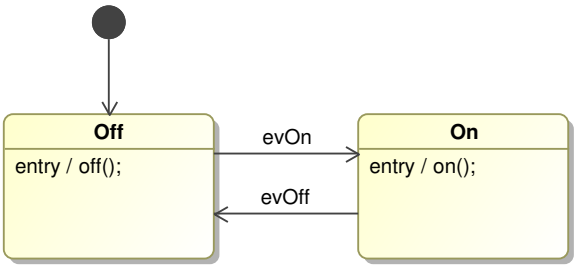


Figure A.4: Indicator Lamp State Machine

APPENDIX

B

# CD-ROM CONTENTS

```
|-- Documentation
|   |-- build
|   |-- images
|   |-- ressources
|   |   `-- Diagrams
|   |-- source
|   |   |-- Analysis
|   |   |   `-- qepVsGenExample
|   |   |-- Appendices
|   |   |   `-- docIncludes
|   |   |-- Conclusion
|   |   |-- Design
|   |   |   `-- docIncludes
|   |   |-- Introduction
|   |   `-- Planning
|   |-- static
|   |   `-- images
|   `-- templates
|-- Papers
`-- Sources
    |-- Examples
    |   |-- CandyMachine
    |   |   |-- Code
    |   |   `-- Model
    |   |-- GenLamp
    |   |   |-- Code
    |   |   `-- Model
    |   `-- QpLamp
    |       |-- Debug
    |       `-- QpLamp
    |-- Externals
    |   `-- QuantumPlatform
    |       `-- qpcpp_4.1.04
    |-- Frameworks
    |   `-- GeneratorQpPort
    |       `-- custom_ports
    |-- Generator
    |   |-- ch.antiserum.scgen
    |   |   |-- bin
    |   |   |-- META-INF
    |   |   |-- model
    |   |   |-- scripts
    |   |   |-- src
```

```
|   |   `-- src-gen
|   `-- ch.antiserum.scgen.beautify
|       |-- bin
|       |-- config
|       |-- META-INF
|       `-- src
|-- Integration
|   `-- MagicDraw
|-- ReferenceImplementations
|   |-- ReferenceModels
|   |   |-- export
|   |   `-- scripts
|   |-- RI1
|   |   `-- RI1
|   `-- RI2
|       |-- Debug
|       `-- RI2
`-- Validation
    |-- Debug
    `-- GeneratorValidation
        |-- Debug
        `-- gen
```

APPENDIX

C

# UTILIZED SOFTWARE PACKAGES

- Eclipse Helios
  - C/C++ Development Tools 7.0.1
  - Eclipse Modeling Tools 1.3.1
  - Graphical Modeling Framework SDK 2.3.0
  - MWE SDK 1.0.0
  - Xpand SDK
  - Xtext DSK
  - Apache Commons Plugin
- Java SE SDK 1.6.0-20
- NoMagic MagicDraw UML 16.5 Standard
- Quantum Leaps Quantum Platform C++ 4.1.04
- Visual Studio 2008 Professional

APPENDIX

D

# NON-PLAGIARISM STATEMENT

I hereby declare that all sources have been mentioned and are referenced according to common scientific rules and practice.

Daniel Michel

# LIST OF FIGURES

# BIBLIOGRAPHY


[Fowler10] Martin Fowler, “Domain Specific Languages”, 23.9.2010

ISBN 978-0-3217-1294-3

Sections 8, 55, 56

[Gronback09] Richard C. Gronback, “Eclipse Modeling Project – A Domain Specific Language (DSL) Toolkit”, 16.3.2009

ISBN 978-0-3215-3407-1

Sections 7 & 14

[Harel87] David Harel, “Statecharts: A Visual Formalism for Complex Systems”, Science of Computer Programming 8, June 1987

http://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/Statecharts.pdf

[MailAbrahamsMsm] Email “Meta State Machine Library Accepted”, From David Abrahams to the Boost mailing list, 12.1.2010

http://lists.boost.org/Archives/boost/2010/01/160812.php

[NoMagic10] No Magic, Inc., “Magic Draw Integrations User Guide”, Version 17.0, 2010

http://www.magicdraw.com/files/manuals/MagicDraw%20Integrations%20UserGuide.pdf

[OMG10] Object Management Group, “The Unified Modeling Language (Superstructure)”, Version 2.3, May 2010

http://www.omg.org/spec/UML/2.3/Superstructure/PDF/

[Samek08] Miro Samek, “Practical UML Statecharts in C/C++”, Second Edition, 1.10.2008

ISBN 978-0-7506-8706-5

Section 4

[WebBoostMsm] “Boost Meta State Machine Documentation”

http://www.boost.org/doc/libs/1_44_0/libs/msm/doc/HTML/index.html

Sections 3 & 4

(retrieved 7.10.2010)

[WebBoostStatechart] “Boost Statechart Documentation”

http://www.boost.org/doc/libs/1_44_0/libs/statechart/doc/index.html

(retrieved 2.10.2010)

[WebEclipseModeling] "Eclipse Modeling Project"

http://www.eclipse.org/modeling/

(retrieved 11.12.2010)

[WebQtStateMachines] "Qt State Machine Framework Documentation"

http://doc.qt.nokia.com/4.7/statemachine-api.html

(retrieved 11.10.2010)

[WebQuantumLeaps] "Quantum Platform State Machine Framework for Embedded Systems"

http://www.state-machine.com/products/index.php

(retrieved 1.10.2010)

[XpandDocs] "Xpand Documentation", Part of the Eclipse Helios Documentation

http://help.eclipse.org/helios/topic/org.eclipse.xpand.doc/help/Reference.html

(retrieved 12.12.2010)